Standard ECMA-234
December 1995

ECMA

Standardizing Information and Communication Systems

Application Programming Interface for Windows

Volume 1

Section 1 - General

Section 2 - Windows Subsystem

Section 3 - Graphics Subsystem

Standard ECMA-234
December 1995

ECMA

Standardizing Information and Communication Systems

Application Programming Interface for Windows

Brief History

The APIW Standard is a functional specification of the Microsoft Windows 3.1 application programming interface. It is based on existing implementations (including Microsoft and others) and behavior. The goal of writing this specification is to define an environment in which:

– applications written to this baseline will be portable to all implementations of the APIW Standard.

– the interface can be enriched through open standards processes to meet current and future user needs in a timely fashion.

APIW uses the current C language binding, and reflects existing coding practices to ensure that current applications will conform to this standard. The APIs documented in this standard shall accurately reflect existing implementations of the windows APIs. If an application that runs with an existing implementation uses one or more APIs contrary to the way it is described in the standard, the standard will be changed to accurately reflect the behavior.

The APIW Standard defines a set of application programming interfaces that allow for the creation of graphical applications spanning a wide range of capabilities. The standard groups these APIs into major functional areas including a window manager interface, a graphics device interface and interfaces necessary for accessing system resources and capabilities. The API requirements of today's major desktop applications are reflected in this specification and are the criteria for determining the APIW content.

The APIW Standard focuses on providing the necessary APIs for writing applications for the desktop, and also allows additional APIs to be bound to an application. This feature enables services outside the scope of a standard desktop application to be provided, for example, database, networking or other system services.

The APIW Standard defines the basic graphical use interface objects, such as buttons, scrollbars, menus, static and edit controls, and the painting functions to draw them, such as area fill, and line and rectangle drawing. Finally, a rich set of text routines in defined, from simple text output to more complex text output routines using multiple founts and font styles, all supporting the use of color.

The APIW Standard is documented in five sections, corresponding loosely to the four functional subsystems represented by the API and the conformance clause. The four APIW sections cover window management, graphical interface, system services and an application support services section. These functions cover window creation and management, graphics routines to paint text and other graphics objects in those windows, functions to access system resources such as files and timers, and finally, common support functions to accelerate the development of graphical window-based applications.

The APIW Window Subsystem section of the standard covers the creation, deletion and management of the window, including window positioning and sizing and the sending and receiving of messages. Within each of these window management subsections are routines that significantly extend the basic functions. With window creation, there are many types of windows that can be created including built-in classes and user-definable classes, that have the ability to modify the style of any one of the built-in classes. Additional functions are defined to affect the display of a window, including functions to modify the windows menu, scrollbars, and the display of carets or cursors within the window. With multiple overlapped windows being displayed simultaneously, functions are defined to manage the position and size of those windows, as well as to control the visibility of a window and its associated icon when it is minimized.

The APIW Window Subsystem section also defines a set of functions for managing a subset of the user interface, referred to as dialog boxes. These functions allow for the creation and management of the dialog box, as well as the user interaction with the dialog box up to its closure. Utility functions are defined to make designing and using a dialog box easier. These utilities provide common dialog box functions, such as group boxes and check boxes, as well as file interface functions to list files and directories. Each of these dialog boxes are controlled by the use of dialog box templates that are stored in resource files.

The APIW Graphics Subsystem section covers all aspects of actually drawing in a window. These aspects include line drawing, text output, graphics primitives, such as rectangles and ellipses, as well as more sophisticated routines such as *floodfill()*, *bitblts()* and *stretchblt()*. The Graphics Device Interface defines bitmaps, icons, cursors and carets, as well as functions to provide for a portable graphics file format called metafiles. The Graphics Device Interface defines a logical coordinate space to further abstract the underlying hardware and has functions to map between the logical and physical coordinate space. The Graphics Device Interface defines utility functions for all drawing routines that use pens, brushes and regions to get precise control over how graphical objects will be drawn.

The APIW System Services section defines platform-independent routines for an application to query the system environment and access system services. System services that may be accessed include memory, timers, the keyboard and the native file system. There are subsections that deal with resources, device I/O and system diagnostic routines. Resource management allows for the loading and unloading of user- and system-defined resources, such as icons, bitmaps and strings. Device I/O includes both parallel and serial port input and output operations. System diagnostic routines enable an application or diagnostic tool to examine the state of an application, including memory utilization, task information and stack usage.

The APIW Application Support Function section defines miscellaneous functions that can be used by a developer in an application. These utility functions define built-in services that a developer does not have to rewrite with each application. These service functions include debugging routines and simple user interface routines to provide graphical feedback to a user. They also include routines for file compression and decompression, standardized routines to retrieve application version information and routines to manage initialization files.

Adopted as an ECMA Standard by the General Assembly of December 1995.

Table of contents

Section 1 - General

1 Scope

This ECMA Standard defines the Windows Application Program Interfaces to the C Programming Language.

2 Conformance

An implementation or application is in full conformance with this standard if all the mandatory requirements of this standard are met.

An implementation or application is in partial conformance with this Standard if a subset of the standard functions is used in conjunction with other windows APIs.

3 References

ISO/IEC/TR 10182:1993,	Information technology - Programming languages, their environments and system software interfaces - Guidelines for language bindings.
ISO 9899:1990,	Programming languages - C.

4 Definitions

For the purposes of this Standard, the following definition applies.

4.1 APIW Standard

A functional specification of the Microsoft Windows 3.1 application programming interface.

5 Formal notations

Two formal notations are used in this ECMA Standard:

- the names of functions and parameters are shown in *italics;*
- the names of structures and elements are shown in **bold**.

Section 2 - Windows Subsystem

6 CallWindowProc

6.1 Synopsis

LRESULT CallWindowProc(WNDPROC wndprcPrev, HWND hWnd, UINT uMsg,

    WPARAM wParam, LPARAM lParam);

6.2 Description

The *CallWindowProc()* function passes message information to the specified window procedure. This function is used for windows subclassing. A subclass is a window or set of windows belonging to the same window class, whose messages are intercepted and processed by another window procedure (or procedures) before being passed to the window procedure of that class.

The parameters are defined as follows:

wndprcPrev	This parameter specifies the address of the previous window procedure.
hWnd	This parameter identifies the window that will receive the message.
uMsg	This parameter specifies the message.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

6.3 Returns

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

6.4 Errors

None.

6.5 Cross-References

SetWindowLong()

7 DispatchMessage

7.1 Synopsis

LONG DispatchMessage(const MSG *lpMsg);

7.2 Description

The *DispatchMessage()* function dispatches a message to a window. The *lpMsg* parameter points to an MSG structure that contains the message. This structure must contain valid message values. If the *lpMsg* parameter points to a WM_TIMER message and the parameter of the WM_TIMER message is not NULL, the *lParam* parameter points to the function that is called instead of the window procedure.

7.3 Returns

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

7.4 Errors

None.

7.5 Cross-References

GetMessage(), PeekMessage(), PostAppMessage(), PostMessage(), TranslateMessage()

8 GetMessage, PeekMessage

8.1 Synopsis

BOOL GetMessage(LPMSG lpMsg, HWND hWnd, UINT uMsgFilterMin,

    UINT uMsgFilterMax);

```
BOOL PeekMessage(LPMSG lpMsg, HWND hWnd, UINT uMsgFilterMin,
        UINT uMsgFilterMax, UINT fuRemove);
```

8.2 Description

The *GetMessage()* function retrieves a message from the application's message queue and places the message in an **MSG** structure. If no message is available, *GetMessage()* yields control to other applications until a message becomes available.

PeekMessage() checks the application's message queue for a message and places the message (if any) in the specified **MSG** structure. Unlike the *GetMessage()* function, the *PeekMessage()* function does not wait for a message to be placed in the queue before returning. It does, however, yield control to other tasks (unless the PM_NOYIELD flag is not set).

The parameters are defined as follows:

lpMsg	This parameter points to the **MSG** structure that is to receive the information about the retrieved message.
hWnd	This parameter identifies the window whose messages are to be retrieved or examined; if this parameter is NULL.
GetMessage or *PeekMessage*	This parameter retrieves messages for any window that belongs to the application making the call.
uMsgFilterMin	This parameter specifies the integer value of the lowest message value to be retrieved or examined.
uMsgFilterMax	This parameter specifies the integer value of the highest message value to be retrieved or examined.
fuRemove (*PeekMessage()* only)	This parameter specifies how messages are handled; this parameter can be a combination of the following values:
PM_NOREMOVE	This parameter does not remove the message from the queue after processing by *PeekMessage()*.
PM_NOYIELD	This parameter prevents the current task from halting and yielding to another task.
PM_REMOVE	This parameter removes the message from the queue after processing by *PeekMessage()*. *GetMessage()* and *PeekMessage()* do not remove WM_PAINT messages from the queue. The messages remain in the queue until processed. *GetMessage()* and *PeekMessage()* retrieve only 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 *uMsgFilterMin* and *uMsgFilterMax* parameters. If *uMsgFilterMin* and *uMsgFilterMax* are both zero, no range filtering is performed.

8.3 Returns

The return value of the GetMessage() function is TRUE if a message other than WM_QUIT is retrieved. The return value is FALSE if a WM_QUIT message is retrieved.

The return value of the *PeekMessage()* function is TRUE if a message is available. Otherwise, it returns FALSE.

8.4 Errors

None.

8.5 Cross-References

IsChild(), PostAppMessage(), SetMessageQueue(), WaitMessage()

9 WaitMessage

9.1 Synopsis

void WaitMessage(void);

9.2 Description

WaitMessage() suspends the application and waits until a new message is added to the application’s queue. If there is a message in the queue that has not previously been examined by a call to *PeekMessage()*, *WaitMessage()* returns immediately. Otherwise, it yields control to other tasks.

9.3 Returns

None.

9.4 Errors

None.

9.5 Cross-References

GetMessage(), PeekMessage()

10 GetMessagePos, GetMessageTime

10.1 Synopsis

DWORD GetMessagePos(void);

LONG GetMessageTime(void);

10.2 Description

The *GetMessagePos()* function returns a long value representing a cursor position, in screen coordinates, when the last message retrieved by the *GetMessage()* function occurs. The x-coordinate is in the low-order word of the return value, the y-coordinate is in the high-order word. The application can use the *MAKEPOINT()* function to obtain a **POINT** structure from the return value.

GetMessageTime() returns the message time for the last message retrieved by the function. The time is a *GetTickCount()* value at the moment the message was placed in the application queue.

10.3 Returns

The *GetMessagePos()* function returns the x- and y- coordinates of the cursor position for the last message, if the function is successful. Otherwise, it returns zero. The *GetMessageTime()* function returns the message time for the last message, if the function is successful. Otherwise, it returns zero.

10.4 Errors

None.

10.5 Cross-References

GetMessage(), GetCursorPos(), MAKEPOINT(), GetTickCount()

11 InSendMessage, ReplyMessage

11.1 Synopsis

BOOL InSendMessage(void);

void ReplyMessage(LRESULT lResult);

11.2 Description

The *InSendMessage()* function determines whether the current window procedure is processing a message sent from another task by a call to the *SendMessage()* function.

The *ReplyMessage()* function is used to reply to a message sent through the *SendMessage()* function without returning control to the application that sent the message. The task that calls *SendMessage()* continues to run as though the task that received the message has returned control. The task that calls *ReplyMessage()* also continues to run. *ReplyMessage()* has no effect if the message was not sent through the *SendMessage()* function or if the message was sent by the same task. The *lResult* parameter of the *ReplyMessage()* function specifies the result of the message processing and is a message-dependent value.

11.3 Returns

The *InSendMessage()* function returns a TRUE value if the window procedure is processing a message sent to it from another task by the *SendMessage()* function. Otherwise, it returns FALSE.

The *ReplyMessage()* function does not return a value.

11.4 Errors

None.

11.5 Cross-References

SendMessage()

---

12 PostAppMessage, PostMessage, PostQuitMessage

12.1 Synopsis

```
typedef struct tagMSG {
        HWND        hWnd;
        UINT        message;
        WPARAM      wParam;
        LPARAM      lParam;
        POINT       pt;
} MSG;

BOOL PostAppMessage(HTASK hTask, UINT uMsg, WPARAM wParam,
        LPARAM lParam);

BOOL PostMessage(HWND hWnd, UINT uMsg, WPARAM wParam,
        LPARAM lParam);

void PostQuitMessage(int nExitCode);
```

12.2 Description

The *PostAppMessage()* function places a message in the message queue of the given task. *PostAppMessage()* then returns without waiting for the task to process the message. When the posted message is retrieved by the application, by calling *GetMessage()* or *PeekMessage()*, the **hwnd** member of the returned **MSG** structure is set to zero.

The parameters of *PostAppMessage()* are defined as follows:

hTask	This parameter specifies the task to which the message is posted.
uMsg	This parameter specifies the message being posted.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

The *PostMessage()* function places a message in a window message queue. *PostMessage()* then returns without waiting for the window to process the message. The parameters of *PostMessage()* are defined as follows:

hWnd	This parameter specifies the window to which the message is to be posted. If this parameter is set to HWND_BROADCAST, the message will be posted to all top-level windows.
uMsg	This parameter specifies the message being posted.

wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

PostQuitMessage() posts a message to the system requesting an application to terminate execution. As a result of this function, the WM_QUIT message is posted to the application, and *PostQuitMessage()* returns immediately.

The *nExitCode* parameter specifies an application-defined exit code, which appears in the *wParam* parameter of the WM_QUIT message posted to the application.

12.3 Returns

The *PostAppMessage()* and the *PostMessage()* functions return a TRUE value in case of success. Otherwise, they return FALSE.

The *PostQuitMessage()* function does not return a value.

12.4 Errors

None.

12.5 Cross-References

GetMessage(), PeekMessage(), SendMessage()

---

13 SendMessage

13.1 Synopsis

LRESULT SendMessage(HWND hWnd, UINT uMsg, WPARAM wParam,

13.2 Description

The *SendMessage()* function sends a message to the specified window or windows. The function calls the window procedure for the window and does not return until that window procedure has processed the message or called *ReplyMessage()*. The parameters of *SendMessage()* are defined as follows:

hWnd	This parameter specifies the window to which the message is to be sent. If this parameter is set to HWND_BROADCAST, the message will be sent to all top-level windows.
uMsg	This parameter specifies the message being sent.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

13.3 Returns

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

13.4 Errors

None.

13.5 Cross-References

InSendMessage(), PostMessage()

---

14 SetMessageQueue

14.1 Synopsis

BOOL SetMessageQueue(int cMsg);

14.2 Description

The *SetMessageQueue()* function is used to set the application's message queue size. The maximum size of the default message queue is eight messages. The call to *SetMessageQueue()* destroys the old message queue along with all the messages it contains. If this function fails, the application has no queue, because the original queue has been deleted before the application attempts to create a new one. In this case the application should call the *SetMessageQueue()* function again, requesting a smaller queue.

The *cMsg* parameter specifies the maximum number of messages that can be contained in the new queue.

14.3 Returns

This function returns TRUE when it is successful. Otherwise, it returns FALSE.

14.4 Errors

None.

14.5 Cross-References

GetMessage(), PeekMessage()

---

15 TranslateAccelerator

15.1 Synopsis

int TranslateAccelerator(HWND hWnd, HACCEL hAccel, MSG *lpMsg);

15.2 Description

The *TranslateAccelerator()* function processes accelerator keys for menu commands. It translates the WM_KEYUP and WM_KEYDOWN messages to WM_COMMAND or WM_SYSCOMMAND messages if there is an entry for the accelerator key in the specified accelerator table. If this function returns a non-zero value (meaning that the message has been translated), the application should not translate the message again by using the *TranslateMessage()* function.

The parameters are defined as follows:

hWnd	This parameter specifies the window where messages are to be translated.
hAccel	This parameter specifies an accelerator table. This value can be obtained by calling the *LoadAccelerators()* function.
lpMsg	This parameter points to a structure containing a message to be translated.

If the call to the *TranslateAccelerator()* results in sending a WM_COMMAND or WM_SYSCOMMAND message, the high-order word of the *lParam* parameter contains the value 1. This differentiates the message from the message sent by menus or controls. The WM_COMMAND or WM_SYSCOMMAND messages are sent, rather than posted. Therefore, *TranslateAccelerator()* does not return until the message is processed.

Accelerator keystrokes that are defined to select items from the system menu are translated into a WM_SYSCOMMAND message. All other accelerator keystrokes are translated into WM_COMMAND messages.

15.3 Returns

This function returns non-zero if the message is translated. Otherwise, it returns zero.

15.4 Errors

None.

15.5 Cross-References

GetMessage(), PeekMessage(), LoadAccelerators()

---

16 TranslateMDISysAccel

16.1 Synopsis

BOOL TranslateMDISysAccel(HWND hWndClient, MSG *lpMsg);

16.2 Description

The *TranslateMDISysAccel()* function processes the accelerator keys of the given multiple document interface (MDI) child window. It translates the WM_KEYUP and WM_KEYDOWN messages to WM_COMMAND or WM_SYSCOMMAND messages. The parameters are defined as follows:

hWndClient	This parameter specifies the parent MDI client window.
lpMsg	This parameter points to a structure containing a message to be translated.

If the call to the *TranslateMDISysAccel()* results in sending the WM_COMMAND or WM_SYSCOMMAND message, the high-order word of the *lParam* parameter contains the value 1, to differentiate the message from messages sent by menus or controls. The WM_COMMAND or WM_SYSCOMMAND message is sent, rather than posted. Therefore, *TranslateMDISysAccel()* does not return until the message is processed.

Accelerator keystrokes that are defined to select items from the system menu are translated into WM_SYSCOMMAND messages. All other accelerator keystrokes are translated into WM_COMMAND messages.

16.3 Returns

This function returns TRUE if it is successful. Otherwise, it returns FALSE.

16.4 Errors

None.

16.5 Cross-References

GetMessage(), PeekMessage()

17 TranslateMessage

17.1 Synopsis

BOOL TranslateMessage(const MSG *lpMsg);

17.2 Description

The *TranslateMessage()* function translates virtual-key messages into character messages. It translates WM_KEYUP and WM_KEYDOWN messages to WM_CHAR or WM_DEADCHAR messages. WM_SYSKEYDOWN and WM_SYSKEYUP messages are translated into WM_SYSCHAR or WM_SYSDEADCHAR messages. The WM_CHAR messages are produced only for the keys mapped to ASCII characters by the keyboard driver. The resulting messages are posted to the application queue. The message structure pointed to by the *lpMsg* parameter is not changed.

The *lpMsg* parameter points to a structure containing a message to be translated. If the application processes virtual-key messages in some other manner (for example, by calling the *TranslateAccelerator()* function), *TranslateMessage()* should not be used.

17.3 Returns

This function returns a TRUE value if the WM_CHAR message is generated as a result of translation. Otherwise, it returns FALSE.

17.4 Errors

None.

17.5 Cross-References

GetMessage(), PeekMessage(), TranslateAccelerator()

18 RegisterWindowMessage

18.1 Synopsis

UINT RegisterWindowMessage(LPCSTR MessageString);

18.2 Description

The *RegisterWindowMessage()* function can be used to create a custom window message or retrieve the identifier associated with an existing custom window message. The *MessageString* parameter is a pointer to a unique string that is associated with the custom message.

The first time the *RegisterWindowMessage()* function is called with a unique message string, a unique message identifier is assigned to the message string and returned. If the *RegisterWindowMessage()* function is called again with the same message string, it will return the message identifier that it used when the window message was first registered.

If the custom message is only used within a single application, the application can forego calling the *RegisterWindowMessage()* function and can use any integer in the range WM_USER through 0x7FFF for the custom window message's identifier.

18.3 Returns

If the *RegisterWindowMessage()* function is successful, it returns an unsigned short integer value that is in the range of 0xC000 through 0xFFFF. If the *RegisterWindowMessage()* function is not successful, it returns zero.

18.4 Errors

None.

18.5 Cross-References

None.

19 GetMessageExtraInfo

19.1 Synopsis

LONG GetMessageExtraInfo(void);

19.2 Description

The *GetMessageExtraInfo()* function gets extra information belonging to the last message found by the *GetMessage()* or *PeekMessage()* functions.

19.3 Returns

If the *GetMessageExtraInfo()* function is successful, it returns a long integer value. The meaning of the return value is understood to the driver calling the function.

19.4 Errors

None.

19.5 Cross-References

None.

20 GetQueueStatus

20.1 Synopsis

DWORD GetQueueStatus(UINT Flags);

20.2 Description

The *GetQueueStatus()* function determines whether any new messages were put in the message queue since the last time the *GetQueueStatus()*, *GetMessage()*, or *PeekMessage()* functions were called. It also retrieves information about the type of messages that are currently in the message queue. The *Flags* parameter defines the message types that should be considered when determining if there are messages currently in the message queue. *Flags* can be one or more of the following values combined using the OR (|) operator:

QS_KEY	This value looks for a WM_CHAR message.
QS_MOUSE	This value looks for a WM_MOUSEMOVE or any mouse button message.
QS_MOUSEMOVE	This value looks for a WM_MOUSEMOVE message.
QS_MOUSEBUTTON	This value looks for any mouse button message.
QS_PAINT	This value looks for a WM_PAINT message.

QS_POSTMESSAGE	This value looks for any message, except for those listed above, that was posted using the *PostMessage()* function.
QS_SENDMESSAGE	This value looks for any message, except for those listed above, that was sent using the *SendMessage()* function.
QS_TIMER	This value looks for a WM_TIMER message.

A message in the message queue can be removed by a system function before the *GetQueueStatus()* function has a chance to return with its information.

20.3 Returns

The *GetQueueStatus()* function returns a DWORD value that holds two types of information. The DWORD's high-order WORD contains the types of messages, combined using the OR (|) operator, that are currently in the message queue. The DWORD's low-order word contains the types of messages, combined using the OR (|) operator, that were put in the message queue since the last time the *GetQueueStatus()*, *GetMessage()*, or *PeekMessage()* functions were called.

20.4 Errors

None.

20.5 Cross-References

GetQueueStatus(), PeekMessage(), PostMessage(), SendMessage()

---

21 RegisterClass, UnregisterClass

21.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT          uStyle;
        WNDPROC       lpfnWndProc;
        int           cbClsExtra;
        int           cbWndExtra;
        HINSTANCE     hInstance;
        HICON         hIcon;
        HCURSOR       hCursor;
        HBRUSH        hBrush;
        LPCSTR        lpszMenuName;
        LPCSTR        lpszClassName;
}WNDCLASS, *LPWNDCLASS;
```

ATOM RegisterClass(const WNDCLASS *lpWndClass);

BOOL UnregisterClass(LPCSTR lpszClassName, HINSTANCE hInstance);

21.2 Description

The *RegisterClass()* function registers a window class for use when creating other windows or class manipulation functions.

The single parameter of this function contains a pointer to a **WNDCLASS** structure, which must be initialized by the caller prior to the call to *RegisterClass()*. Information contained in this structure is stored internally and used in subsequent window creation calls to set various properties of windows belonging to this class.

The CS_GLOBALCLASS bit in the **uStyle** member of the **WNDCLASS** structure defines whether the class being created is a task-specific or application-global class. Registering an application-global class allows creation of windows of this class that are owned by *hInstance* parameters other than the one used in class registration.

If an application attempts to register a global class with a name that is already registered, *RegisterClass()* fails. An application cannot register two task-specific classes with the same name. However, overriding the global class with a task-specific class with the same name is possible.

The *UnregisterClass()* function removes a previously registered window class, freeing the memory associated with the class. The **lpszClassName** member specifies the window class name, and can be a pointer to a zero-terminated string, or an integer value. If the integer value is used, it is stored in the lower-order word of **lpszClassName**. The high-order word in this case must be zero. The MAKEINTRESOURCE macro can be used to generate such a value.

The *hInstance* parameter identifies the owner of the class (the task that created the class). If the class with the specified name cannot be found, the *UnregisterClass()* function fails.

If a window created with the specified class still exists, the *UnregisterClass()* function fails.

System global classes, such as buttons or list boxes, cannot be unregistered.

21.3 Returns

The *RegisterClass()* function returns an atom uniquely identifying the class being registered if the function is successful. Otherwise, it returns FALSE. The *UnregisterClass()* function returns TRUE if it is successful. Otherwise, it returns FALSE.

21.4 Errors

None.

21.5 Cross-References

GetClassInfo(), GetClassName(), GetClassLong(), GetClassWord(), SetClassLong(), SetClassWord(), CreateWindow(), CreateWindowEx(), MAKEINTRESOURCE

---

22 GetClassInfo

22.1 Synopsis

```
typedef struct tagWNDCLASS {
            UINT        uStyle;
            WNDPROC     lpfnWndProc;
            int         cbClsExtra;
            int         cbWndExtra;
            HINSTANCE   hInstance;
            HICON       hIcon;
            HCURSOR     hCursor;
            HBRUSH      hBrush;
            LPCSTR      lpszMenuName;
            LPCSTR      lpszClassName;
} WNDCLASS, *LPWNDCLASS;

BOOL GetClassInfo(HINSTANCE hInstance, LPCSTR lpszClassName,
            LPWNDCLASS lpWndClass);
```

22.2 Description

The *GetClassInfo()* function retrieves information about a previously registered window class.

The *lpszClassName* parameter specifies the window class name, and can be a pointer to a zero-terminated string, or an integer value. If the integer value is used, it is stored in the lower-order word of the *lpszClassName*. The high-order word in this case must be zero. The MAKEINTRESOURCE macro can be used to generate this value.

The *hInstance* parameter specifies the instance of the application that created the class. To retrieve information about a system global class, such as dialog class, this parameter must be set to zero.

The *lpWndClass* parameter points to the **WNDCLASS** structure that receives the information about the class. The *GetClassInfo()* function does not fill the **lpszMenuName** and **lpszClassName** fields of the **lpWndClass** structure.

If the class with the specified name owned by the specified instance cannot be found, *GetClassInfo()* fails.

22.3 Returns

GetClassInfo() returns TRUE if it is successful. Otherwise, it returns FALSE.

22.4 Errors

None.

22.5 Cross-References

RegisterClass(), UnregisterClass(), GetClassName(), GetClassLong(), GetClassWord(), SetClassLong(), SetClassWord(), CreateWindow(), CreateWindowEx(), MAKEINTRESOURCE

---

23 GetClassName

23.1 Synopsis

int GetClassName(HWND hWnd, LPSTR lpszClassName, int cchClassName);

23.2 Description

The *GetClassName()* function retrieves the class name for a given window. The *hWnd* parameter specifies the window to be retrieved. The *lpszClassName* parameter points to a buffer to be filled with the zero-terminated class name string.

The *cchClassName* parameter specifies the length of the buffer pointed to by *lpszClassName*. If the retrieved class name string is longer than *cchClassName* - 1, it is truncated to fit the buffer. If the window handle in *hWnd* is invalid, *GetClassName()* fails.

23.3 Returns

GetClassName() returns the length, in bytes, of the returned class name without the terminating zero, if it is successful. Otherwise, it returns zero.

23.4 Errors

None.

23.5 Cross-References

GetClassInfo()

---

24 GetClassWord, SetClassWord

24.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT        uStyle;
        WNDPROC     lpfnWndProc;
        int         cbClsExtra;
        int         cbWndExtra;
        HINSTANCE   hInstance;
        HICON       hIcon;
        HCURSOR     hCursor;
        HBRUSH      hBrush;
        LPCSTR      lpszMenuName;
        LPCSTR      lpszClassName;
} WNDCLASS, *LPWNDCLASS;
```

WORD GetClassWord(HWND hWnd, int nIndex);

WORD SetClassWord(HWND hWnd, int nIndex, WORD wNewWord);

24.2 Description

The *GetClassWord()* and *SetClassWord()* functions retrieve and set a word value at the specified offset from extra memory associated with the class associated with the specified window. Extra class memory is allocated by specifying the desired number of bytes in the **cbClsExtra** member of the **WNDCLASS** structure when the class is registered via the *RegisterClass()* function. The *hWnd* parameter identifies the window whose class is to be used for retrieving or setting the word value.

The *nIndex* parameter specifies the zero-based offset of the word value relative to the base of the class extra memory. If *nIndex* is greater than or equal to zero, the valid range is from zero to the amount of extra memory for the class in bytes, minus the size of the WORD in bytes.

There are several negative values used to retrieve or set the following class components:

GCW_HBRBACKGROUND	This value handles of the brush associated with the class.
GCW_HCURSOR	This value handles of the cursor associated with the class.
GCW_HICON	This value handles of the icon associated with the class.
GCW_HMODULE	This value handles of the module that registered the class.
GCW_CBCLSEXTRA	This value specifies the size, in bytes, of the extra memory associated with the class. Setting this value does not change the amount of memory already allocated.
GCW_CBWNDEXTRA	This value specifies the size, in bytes, of the extra memory associated with each window of the class. Setting this value does not change the amount of memory already allocated.
GCW_STYLE	This value specifies the window class style bits.
GCW_ATOM	This value specifies the atom for the class name string or integer value used with the *RegisterClass()* function call.

The *wNewWord* parameter of the *SetWindowWord()* function specifies the new word value.

24.3 Returns

GetWindowWord() returns the retrieved word value, if the function is successful. Otherwise, it returns zero. *SetWindowWord()* returns the previous word value at the specified offset, if the function is successful. Otherwise, it returns zero.

24.4 Errors

None.

24.5 Cross-References

SetClassLong(), GetClassLong(), RegisterClass(), SetWindowWord(), GetWindowWord()

25 GetClassLong, SetClassLong

25.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT        uStyle;
        WNDPROC     lpfnWndProc;
        int             cbClsExtra;
        int             cbWndExtra;
        HINSTANCE       hInstance;
        HICON           hIcon;
        HCURSOR         hCursor;
        HBRUSH          hBrush;
        LPCSTR          lpszMenuName;
        LPCSTR          lpszClassName;
} WNDCLASS, *LPWNDCLASS;
```

DWORD GetClassLong(HWND hWnd, int nIndex);

DWORD SetClassLong(HWND hWnd, int nIndex, DWORD dwNewLong);

25.2 Description

The *GetClassLong()* and *SetClassLong()* functions retrieve and set a long value at the specified offset from the extra memory associated with the specified window. Extra class memory is allocated by specifying a desired number of bytes in the **cbClsExtra** member of the **WNDCLASS** structure, when the class is registered via the *RegisterClass()* function. The *hWnd* parameter identifies the window class that is used for retrieving or setting the long value. The *nIndex* parameter specifies the zero-based offset of the long value relative to the base of the class extra memory.

If *nIndex* is greater than or equal to zero, the valid range is zero to *n*, where *n* equals the amount of extra memory for the class, in bytes, minus the size of the DWORD**,** in bytes.

There are several negative values used to retrieve or set the following class components:

GCL_MENUNAME	This value specifies the address of the menu name string, where the string identifies the menu associated with the class.
GCL_WNDPROC	This value specifies the address of the window procedure associated with the class.

The *dwNewLong* parameter of the *SetWindowLong()* function specifies the new long value.

25.3 Returns

GetWindowLong() returns the retrieved long value if the function is successful. Otherwise, it returns a zero. *SetWindowLong()* returns the previous long value at the specified offset, if the function is successful. Otherwise, it returns a zero.

25.4 Errors

None.

25.5 Cross-References

SetClassWord(), GetClassWord(), RegisterClass(), SetWindowLong(), GetWindowLong(), GetClassInfo()

---

26 SetWindowLong

26.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT            uStyle;
        WNDPROC         lpfnWndProc;
        int             cbClsExtra;
        int             cbWndExtra;
        HINSTANCE       hInstance;
```

HICON	**hIcon;**
HCURSOR	**hCursor;**
HBRUSH	**hBrush;**
LPCSTR	**lpszMenuName;**
LPCSTR	**lpszClassName;**

} WNDCLASS, *LPWNDCLASS;

LONG SetWindowLong(HWND hWnd, int ByteOffset, LONG Value);

26.2 Description

SetWindowLong() sets a value of LONG in a window's extra memory. The *hWnd* parameter is the handle to a window that has extra memory. The *Value* parameter contains the LONG value to store in the window's extra memory. The *ByteOffset* parameter contains the LONG value's byte position in the window's extra memory. A zero value sets the value of the first LONG in the extra memory. A value of 4 sets the value of the second LONG, and so on. The *ByteOffset* parameter can be used to retrieve standard information about the window, and can also be one of the following values:

GWL_EXSTYLE	This value sets the window's extended window style
GWL_STYLE	This value sets the window's window style. Do not use this flag to change the window's WS_DISABLE style; instead, call the *EnableWindow()* function to change the window's disabled state.
GWL_WNDPROC	This value sets a long pointer to the window's window procedure.

If the window is a dialog box, the *ByteOffset* parameter can be one of the following values:

DWL_DLGPROC	This value sets the address of the window's dialog box procedure.
DWL_MSGRESULT	This value sets the return value of a message that the dialog box procedure processed.
DWL_USER	This value sets the application's private extra information.

A window class is created by describing the class in a **WNDCLASS** structure and passing the structure to the *RegisterClass()* function. A window class can be defined as having extra memory by placing a positive value in the **WNDCLASS** structure's **cbWndExtra** element. The value of the **cbWndExtra** element represents the number of bytes of extra memory to be allocated for each window belonging to the class.

26.3 Returns

If the *SetWindowLong()* function is successful, it returns the previous LONG value that was located at the specified byte position in the window's extra memory. If the *SetWindowLong()* function is not successful, it returns zero.

26.4 Errors

None.

26.5 Cross-References

GetWindowLong(), RegisterClass(), **WNDCLASS**

27 CreateWindow, CreateWindowEx

27.1 Synopsis

typedef struct tagWNDCLASS {

UINT	**uStyle;**
WNDPROC	**lpfnWndProc;**
int	**cbClsExtra;**
int	**cbWndExtra;**

```
        HINSTANCE  hInstance;
        HICON      hIcon;
        HCURSOR    hCursor;
        HBRUSH     hBrush;
        LPCSTR     lpszMenuName;
        LPCSTR     lpszClassName;
} WNDCLASS, *LPWNDCLASS;
typedef struct tagCREATESTRUCT {
        LPVOID     lpCreateParams;
        HINSTANCE  hInstance;
        HMENU      hMenu;
        HWND       hWndParent;
        int        cy;
        int        cx;
        int        y;
        int        x;
        LONG       dwStyle;
        LPCSTR     pszName;
        LPCSTR     pszClass;
        DWORD      dwExStyle;
} CREATESTRUCT, *LPCREATESTRUCT;
HWND CreateWindow(LPCSTR lpszClassName, LPCSTR lpszWindowName, DWORD dwStyle,
        int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
        LPVOID lpCreateParams);
HWND CreateWindowEx(DWORD dwExStyle, LPCSTR lpszClassName, LPCSTR, lpszWindowName,
        DWORD dwStyle, int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu,
        HINSTANCE hInstance, LPVOID lpCreateParams);
```

27.2 Description

The *CreateWindow()* and *CreateWindowEx()* functions are used to make a window. *CreateWindowEx()* differs from *CreateWindow()* by one extra parameter - extended window style.

The description of the parameters follows:

dwExStyle (*CreateWindowEx()* only
: This parameter specifies the extended style of the window, and can be one of the following values:

WS_EX_DLGMODALFRAME
: This parameter creates a window with a double border.

WS_EX_NOPARENTNOTIFY
: This parameter suppresses WM_PARENTNOTIFY messages from the child window to its ancestors upon window creation or destruction.

WS_EX_TOPMOST
: This parameter specifies that the window being created should be placed above all non-topmost windows and stay above them even when

	deactivated. This is equivalent to calling the *SetWindowPos()* parameter with the HWND_TOPMOST parameter. This attribute can be removed by calling the *SetWindowPos()* parameter with the HWND_NOTOPMOST parameter.
WS_EX_ACCEPTFILES	This parameter indicates that the window accepts drag-drop files.
WS_EX_TRANSPARENT	This parameter creates a transparent window. Such a window does not obscure windows beneath it.
lpszClassName	This parameter specifies a class name used for the window; it is a pointer to a zero-terminated string containing a class name, or an integer value, in which case the high-order word must be zero. The class requested must have been previously registered, or must be one of the system pre-defined classes:
	BUTTON
	COMBOBOX
	DIALOG (or WC_DIALOG value)
	EDIT
	LISTBOX
	MDICLIENT
	SCROLLBAR
	STATIC
lpszWindowName	This parameter specifies a window name. This parameter is a pointer to a zero-terminated string containing a window name, or zero if an empty name is to be used. If the window has the WS_CAPTION style of *dwStyle* set, the specified name (if non-empty) appears in the window's title bar. If the window being created is a control (such as button, static control or edit control), *lpszWindowName* is used to specify the text of the control.
dwStyle	This parameter specifies the style mask of the window being created. It can be a combination of the window styles described below, and the control-specific styles. The valid window styles are:
WS_BORDER	This parameter creates a window with a border.
WS_CAPTION	This parameter creates a window with a title bar. The WS_BORDER style is implied. This parameter cannot be combined with the WS_DLGFRAME style.
WS_CHILD	This parameter creates a child window. If combined with the WS_POPUP style, WS_POPUP takes precedence.
WS_CHILDWINDOW	Same as WS_CHILD.
WS_CLIPCHILDREN	This parameter excludes the child windows areas when drawing within the parent window.
WS_CLIPSIBLINGS	This parameter clips siblings relative to each other. If not specified, it is possible while drawing in the window's client area to draw into the area of the overlapping sibling. This style is always turned on for non-pop-up top-level windows.
WS_DISABLED	This parameter creates a window that is initially disabled.
WS_DLGFRAME	This parameter creates a window with a double border and no title.

WS_GROUP	This parameter specifies the first control of a group of controls in which the user can move the input focus using the arrow keys. The next control created with this style ends the current group and starts the next group.
WS_HSCROLL	This parameter creates a window that has a horizontal scroll bar.
WS_MAXIMIZE	This parameter creates a window that is initially maximized.
WS_MAXIMIZEBOX	This parameter creates a window that has a Maximize button.
WS_MINIMIZE	This parameter creates a window that is initially minimized.
WS_MINIMIZEBOX	This parameter creates a window that has a Minimize button.
WS_OVERLAPPEDWINDOW	This parameter combines the WS_OVERLAPPED (a placeholder), WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	This parameter creates a pop-up window (takes precedence over the WS_CHILD style).
WS_POPUPWINDOW	This parameter combines WS_POPUP, WS_BORDER and WS_SYSMENU styles. The system menu is visible only if the WS_CAPTION style is also specified.
WS_SYSMENU	This parameter creates a window that has a system-menu box on its title bar.
WS_TABSTOP	This parameter specifies that the control being created belongs to the group in which the user can move the input focus using the TAB key.
WS_THICKFRAME	This parameter creates a window with a thick frame that can be used to resize a window using a mouse.
WS_VISIBLE	This parameter creates a window that is initially visible.
WS_VSCROLL	This parameter creates a window that has a vertical scroll bar.

x	This parameter specifies the initial x-coordinate of the window, and is an x-coordinate of the window's upper-left corner in the parent's coordinate system for the child windows or in the screen coordinate system for the top-level windows. If the special value CW_USEDEFAULT is passed, the system sets the default position and ignores the *y* parameter for overlapped windows; for non-overlapped windows *x* and *y* parameters are set to zero.
y	This parameter specifies the initial y-coordinate of the window, and is a y-coordinate of the window's upper-left corner in the parent's coordinate system for the child windows or in the screen coordinate system for the top-level windows. If the window being created is a list box control, *y* is the y-coordinate of the upper-left corner of the control's client area; if the window being created overlaps the window, and the *x* parameter is specified as CW_USEDEFAULT, the *y* parameter is ignored.
nWidth	This parameter specifies the width of the window in device units. If the special value CW_USEDEFAULT is passed, the system sets the default width and height of the window and ignores the *nHeight* parameter for overlapped windows; for non-overlapped windows, the *nWidth* and *nHeight* parameters are set to zero.
nHeight	This parameter specifies the height of the window in device units. If the window being created is an overlapped window, and the *nWidth* parameter is specified as CW_USEDEFAULT, the *nHeight* parameter is ignored.

hWndParent	This parameter specifies the window handle of the parent or owner window. It must be a valid window handle or HWND_DESKTOP to indicate that the window does not have a parent window or is not owned by another window. An owned window is a top-level window that is destroyed when its owner window is destroyed, hidden when its owner is minimized, and is always displayed on top of its owner window.
hMenu	This parameter specifies the menu used with the window (for top-level windows), or the child ID (for child windows). The child ID uniquely identifies the child window within the set of the child windows with the same parent, and is used by controls to notify their parent of events (such as button clicks). Passing a value of zero for the top-level window requests the use of the class menu (as defined by the **lpszMenuName** member of the **WNDCLASS** structure).
hInstance	This parameter specifies the instance of the module that owns the window. This value is used by the system to access resources necessary for the window, such as menu.
lpCreateParams	This parameter specifies the value passed to the window procedures in the **lpCreateParams** member of the **CREATESTRUCT** structure when WM_NCCREATE and WM_CREATE messages are sent. If the window being created is of the class MDICLIENT, **lpCreateParams** must point to the CLIENTCREATESTRUCT structure.

During window creation, the following messages are sent to the window procedure:

WM_GETMINMAXINFO (optional, governed by the window style)

WM_NCCREATE

WM_NCCALCSIZE

WM_CREATE

If the WS_VISIBLE bit of the dwStyle parameter is set, the created window is shown and activated, which sends additional messages to the window procedure.

If the window does not have the WS_EX NOPARENTNOTIFY bit in the extended style set, and is created as a child window, WM_PARENTNOTIFY messages are sent to the window's ancestors during its creation.

27.3 Returns

This function returns the value of the handle of the new window when it is successful. Otherwise, it returns zero.

27.4 Errors

None.

27.5 Cross-References

RegisterClass(), SetWindowPos()

---

28 DestroyWindow

28.1 Synopsis

BOOL DestroyWindow(HWND hWnd);

28.2 Description

The *DestroyWindow()* function destroys the specified window and all its children, if any. If the window being destroyed is a top-level window, *DestroyWindow()* also destroys all top-level windows owned by this window. The function destroys owned windows first, then child windows, then the window itself.

During the destruction process *DestroyWindow()* sends messages to all windows being destroyed to deactivate them and remove the input focus. It also destroys the menu and/or the system menu associated with a window, destroys window timers and carets, disowns the clipboard, and breaks the clipboard-viewer chain (if the window is on the top of this chain). The application's message queue is flushed to remove messages for the windows being destroyed.

DestroyWindow() also sends WM_DESTROY and WM_NCDESTROY messages to the window and notifies ancestors of child windows via a WM_PARENTNOTIFY message, if the window does not have the WS_EX_NOPARENTNOTIFY style bit set.

The single parameter of this function specifies the handle of the window to be destroyed. If this handle is invalid, *DestroyWindow()* fails.

28.3 Returns

The function returns TRUE is the function is successful. Otherwise, it returns FALSE.

28.4 Errors

None.

28.5 Cross-References

CreateWindow(), CreateWindowEx()

---

29 WindowProc

29.1 Synopsis

LRESULT WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

29.2 Description

The *WindowProc()* function is an application-defined callback procedure that processes messages sent to the window. This name is a placeholder for the real function defined by the window's creator. When the window is being created, the system uses the class procedure as the initial value. The address of the window procedure is changed by the application, which calls *SetWindowLong()* with GLW_WNDPROC and a new value.

The parameters are defined as follows:

hWnd	This parameter identifies the window to which the message is being sent.
uMsg	This parameter specifies the message being sent.
wParam	This parameter specifies the additional message-dependent information.
lParam	This parameter specifies the additional message-dependent information.

29.3 Returns

The return value is the result of the processing of the message.

29.4 Errors

None.

29.5 Cross-References

CreateWindow(), CreateWindowEx(), RegisterClass(), SetWindowLong()

---

30 DefDlgProc

30.1 Synopsis

LRESULT DefDlgProc(HWND hWndDlg, UINT uMsg, WPARAM wParam, LPARAM lParam);

30.2 Description

The *DefDlgProc()* function performs default processing for any messages that a dialog box, with an application-defined window class, does not process. This function is the window procedure for the system dialog class. If the application is creating a dialog box with an application-defined class, it can use this function instead of *DefWindowProc()* to carry out default message processing.

DefDlgProc() will first call the application-defined dialog box procedure (if any), and then perform default processing if this procedure returns FALSE.

DefDlgProc() must not be called by a dialog box procedure. Doing so results in recursive execution.

The parameters are defined as follows:

hWndDlg	This parameter identifies the dialog box.
uMsg	This parameter specifies the message.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

30.3 Returns

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

30.4 Errors

None.

30.5 Cross-References

DefWindowProc()

---

31 DefFrameProc

31.1 Synopsis

LRESULT DefFrameProc(HWND hWnd, HWND hWndMDIClient, UINT uMsg, WPARAM wParam,

LPARAM lParam);

31.2 Description

The *DefFrameProc()* function performs default processing for any messages that the window procedure of an MDI frame window does not process. All messages that are not explicitly processed by the window procedure must be passed to *DefFrameProc()* function, not to the *DefWindowProc()* function. If the *hWndMDIClient* parameter is zero, the *DefFrameProc()* passes the message to *DefWindowProc()*.

The parameters are defined as follows:

hWnd	This parameter identifies the MDI frame window.
hWndMDIClient	This parameter identifies the MDI client window.
uMsg	This parameter specifies the message.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

31.3 Returns

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

31.4 Errors

None.

31.5 Cross-References

DefWindowProc(), DefMDIChildProc()

---

32 DefMDIChildProc

32.1 Synopsis

LRESULT DefMDIChildProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

32.2 Description

The *DefMDIChildProc()* function performs default processing for any messages that the window procedure of an MDI child window does not process. All messages that are not explicitly processed by the window procedure must be passed to the *DefMDIChildProc()* function, not to the *DefWindowProc()* function.

The parameters are defined as follows:

hWnd	This parameter identifies the MDI child window. It is assumed that the parent of this window is of the MDICLIENT class.
uMsg	This parameter specifies the message.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

32.3 Returns

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

32.4 Errors

None.

32.5 Cross-References

DefWindowProc(), DefFrameProc()

---

33 DefWindowProc

33.1 Synopsis

LRESULT DefWindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

33.2 Description

The *DefWindowProc()* function performs default processing for any messages that an application-defined window procedure does not process. It should be called when an application wants the system to carry out the default processing for the message.

The parameters are defined as follows:

hWnd	This parameter identifies the window.
uMsg	This parameter specifies the message.
wParam	This parameter specifies additional message-dependent information.
lParam	This parameter specifies additional message-dependent information.

33.3 Returns

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

33.4 Errors

None.

33.5 Cross-References

DefDlgProc()

---

34 GetWindowWord

34.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT          uStyle;
        WNDPROC       lpfnWndProc;
        int           cbClsExtra;
        int           cbWndExtra;
        HINSTANCE     hInstance;
        HICON         hIcon;
        HCURSOR       hCursor;
        HBRUSH     hBrush;
        LPCSTR     lpszMenuName;
        LPCSTR     lpszClassName;
} WNDCLASS, *LPWNDCLASS;
```

WORD GetWindowWord(HWND hWnd, int ByteOffset);

34.2 Description

The *GetWindowWord()* function returns a WORD value from a window's extra memory. The *hWnd* parameter is the handle to a window that has extra memory. The *ByteOffset* parameter contains the WORD value's byte position in the window's extra memory. A zero value retrieves the first WORD in the extra memory, a value of 2 retrieves the second WORD, and so on.

The *ByteOffset* parameter can also be used to retrieve standard information about the window, and can be one of the following values:

GWW_HINSTANCE	This value retrieves the instance handle of the module that owns the window.
GWW_HWNDPARENT	This value retrieves the window handle of the parent window of *hWnd*.
GWW_ID	This value retrieves the window's child window identifier.

A window class is created by describing the class in a **WNDCLASS** structure and passing the structure to the *RegisterClass()* function. A window class can be defined as having extra memory by placing a positive value in the **WNDCLASS** structure's **cbWndExtra** element. The value of the **cbWndExtra** element represents the number of bytes of extra memory to allocate for each window belonging to the class.

34.3 Returns

If the *GetWindowWord()* function is successful, it returns the desired WORD value.

34.4 Errors

None.

34.5 Cross-References

RegisterClass(), SetWindowWord(), **WNDCLASS**

---

35 GetWindowLong

35.1 Synopsis

```
typedef struct tagWNDCLASS {
        UINT       uStyle;
        WNDPROC    lpfnWndProc;
        int        cbClsExtra;
        int        cbWndExtra;
        HINSTANCE  hInstance;
        HICON      hIcon;
        HCURSOR    hCursor;
        HBRUSH     hBrush;
        LPCSTR     lpszMenuName;
        LPCSTR     lpszClassName;
} WNDCLASS, *LPWNDCLASS;
```

LONG GetWindowLong(HWND hWnd, int ByteOffset);

35.2 Description

The *GetWindowLong()* function returns a LONG value from a window's extra memory. The *hWnd* parameter is the handle to a window that has extra memory. The *ByteOffset* parameter contains the LONG value's byte position in the window's extra memory. A zero value retrieves the first LONG in the extra memory, a value of 4 retrieves the second LONG, and so on.

The *ByteOffset* parameter can be used to retrieve standard information about the window, and can be one of the following values:

GWL_EXSTYLE	This value retrieves the window's extended window style.
GWL_STYLE	This value retrieves the window's window style.
GWL_WNDPROC	This value retrieves a long pointer to the window's window procedure.

If the window is a dialog box, the *ByteOffset* parameter can be one of the following values:

DWL_DLGPROC	This value retrieves the address of the dialog box's window procedure.
DWL_MSGRESULT	Some messages that are sent to a dialog box use its BOOL return value to return non-BOOL values; use the DWL_MSGRESULT flag to retrieve the return values of the following dialog box messages: WM_CHARTOITEM WM_COMPAREITEM WM_CTLCOLOR WM_INITDIALOG WM_QUERYDRAGICON WM_VKEYTOITEM
DWL_USER	This value retrieves the application's private extra information.

A window class is created by describing the class in a **WNDCLASS** structure and passing the structure to the *RegisterClass()* function. A window class can be defined as having extra memory by placing a positive value in the **WNDCLASS** structure's **cbWndExtra** element. The value of the **cbWndExtra** element represents the number of bytes of extra memory to allocate for each window belonging to the class.

35.3 Returns

If the *GetWindowLong()* function is successful, it returns the desired LONG value.

35.4 Errors

None.

35.5 Cross-References

RegisterClass(), SetWindowLong(), **WNDCLASS**

---

36 SetWindowWord

36.1 Synopsis

typedef struct tagWNDCLASS {

UINT	**uStyle;**
WNDPROC	**lpfnWndProc;**
int	**cbClsExtra;**
int	**cbWndExtra;**
HINSTANCE	**hInstance;**
HICON	**hIcon;**

HCURSOR **hCursor;**

HBRUSH **hBrush;**

LPCSTR **lpszMenuName;**

LPCSTR **lpszClassName;**

} WNDCLASS, *LPWNDCLASS;

WORD SetWindowWord(HWND hWnd, int ByteOffset, WORD Value);

36.2 Description

The *SetWindowWord()* function sets the value of a WORD in a window's extra memory. The *hWnd* parameter is the handle to a window that has extra memory. The *Value* parameter contains the value of the WORD to store in the window's extra memory. The *ByteOffset* parameter contains the byte position in the window's extra memory. A zero value sets the value of the first WORD in the extra memory, a value of 2 sets the value of the second WORD, and so on.

ByteOffset can be used to set standard information about the window, and can also be one of the following values:

GWW_HINSTANCE	This value retrieves the instance handle of the module that owns the window.
GWW_ID	This value retrieves the window's child window identifier.

A window class is created by describing the class in a WNDCLASS structure and passing the structure to the *RegisterClass()* function. A window class can be defined as having extra memory by placing a positive value in the **WNDCLASS** structure's **cbWndExtra** element. The value of the **cbWndExtra** element represents the number of bytes of extra memory to allocate for each window belonging to the class.

36.3 Returns

If the *SetWindowWord()* function is successful, it returns the previous WORD value located at the specified byte position in the window's extra memory. If the *SetWindowWord()* function is not successful, it returns zero.

36.4 Errors

None.

36.5 Cross-References

GetWindowWord(), RegisterClass(), **WNDCLASS**

37 BeginDeferWindowPos

37.1 Synopsis

HDWP BeginDeferWindowPos(int cWindows);

37.2 Description

The *BeginDeferWindowPos()* function allocates a storage area for window-positioning information. The *DeferWindowPos()* function fills this storage area with information for a window that is about to be moved, resized, shown, hidden, or activated. The *EndDeferWindowPos()* function uses the window-positioning information stored in the buffer to update the position of the windows.

The *cWindows* parameter specifies the initial number of window-positioning requests to be stored in the structure being created. The system can expand the structure beyond its initial size during the subsequent *DeferWindowPos()* calls. However, if the memory allocation required for the expansion fails, the entire window-positioning sequence fails.

37.3 Returns

If the function is successful, a handle to a deferred window position structure is returned. If the function is not successful, the value NULL is returned.

37.4 Errors

None.

37.5 Cross-References

DeferWindowPos(), EndDeferWindowPos()

---

38 EndDeferWindowPos

38.1 Synopsis

BOOL EndDeferWindowPos(HDWP hDWP);

38.2 Description

The *EndDeferWindowPos()* function uses the window-positioning information stored in the buffer to update the position of the windows.

The *BeginDeferWindowPos()* function allocates a storage area for window-positioning information. The *DeferWindowPos()* function fills this storage area with information for a window that is about to be moved, resized, shown, hidden, or activated.

The *hDWP* parameter is a handle to a deferred window position structure that was returned by the *BeginDeferWindowPos()* function.

38.3 Returns

If the function is successful, the value TRUE is returned. If the function is not successful, the value FALSE is returned.

38.4 Errors

None.

38.5 Cross-References

BeginDeferWindowPos(), DeferWindowPos()

---

39 DeferWindowPos

39.1 Synopsis

HDWP DeferWindowPos(HWDP hDWP, HWND hWnd, HWND hWndInsertAfter, int x, int y, int cx,

int cy, UINT uiFlags);

39.2 Description

The *DeferWindowPos()* function fills a storage area with information for a window that is about to be moved, resized, shown, hidden, or activated.

The *BeginDeferWindowPos()* function allocates this storage area for window-positioning information. The *EndDeferWindowPos()* function uses the window-positioning information stored in the buffer to update the position of the windows.

The *DeferWindowPos()* function has the following parameters:

hDWP	This parameter identifies the internal structure used as storage for the window-positioning request. This value is returned by the *BeginDeferWindowPos()* function or by the most recent call to the *DeferWindowPos()* function.
hWnd	This parameter identifies the window where the window-positioning request is to be stored.
hWndInsertAfter	This parameter identifies the window that precedes the positioned window in the stacking order (Z-order); it must be a valid window handle, or one of the following values:

HWND_BOTTOM	This parameter places the window at the bottom of the Z-order; if *hWnd* is a topmost window, it will lose its topmost status.
HWND_TOP	This parameter places the window at the top of the Z-order.
HWND_TOPMOST	This parameter places the window above all non-topmost windows; the window maintains its topmost position even if deactivated.
HWND_NOTOPMOST	This parameter places the window at the top of all non-topmost windows (behind all topmost windows), thus resetting the window's topmost state.
x	The screen position of the left side of the window.
y	The screen position of the top side of the window.
cx	The width of the window.
cy	The height of the window.
uiFlags	Window size and position options. One or more of the following constant values can be OR'ed together:
SWP_DRAWFRAME	Draws a window class's frame around the window.
SWP_HIDEWINDOW	Causes the window to be hidden.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Does not change the window's current position.
SWP_NOSIZE	Does not change the window's current size.
SWP_NOREDRAW	Does not redraw changes. If this flag is set, no part of the window or parent window is redrawn. The application must invalidate the windows' areas and redraw them.
SWP_NOZORDER	Does not change the window ordering.
SWP_SHOWWINDOW	Caused the window to be shown.

39.3 Returns

If the function is successful, a new handle to the deferred window position structure is returned. This new handle should replace the handle passed to the *DeferWindowPos()* function and should be used in future function calls to *DeferWindowPos()* and *BeginDeferWindowPos()*. If the function is not successful, the value NULL is returned.

39.4 Errors

None.

39.5 Cross-References

BeginDeferWindowPos(), EndDeferWindowPos()

---

40 SetWindowPos

40.1 Synopsis

BOOL SetWindowPos(HWND hWnd, HWND hWndInsertAfter, int x, int y, int cx, int cy, UINT uiFlags);

40.2 Description

The *SetWindowPos()* function executes a single window-positioning request. Calling this function is equivalent to calling:

EndDeferWindowPos(DeferWindowPos(BeginDeferWindowPos(1),hWnd,hWndInsertAfter,x,y,

cx,cy,uiFlags))

The *SetWindowPos()* function has the following parameters:

hWnd	This parameter identifies the window where the window-positioning request is to be stored.
hWndInsertAfter	This parameter identifies the window that precedes the positioned window in the stacking order (Z-order); it must be a valid window handle, or one of the following values:
HWND_BOTTOM	This parameter places the window at the bottom of the Z-order; if *hWnd* is a topmost window, it will lose its topmost status.
HWND_TOP	This parameter places the window at the top of the Z-order.
HWND_TOPMOST	This parameter places the window above all non-topmost windows; the window maintains its topmost position even if deactivated.
HWND_NOTOPMOST	This parameter places the window at the top of all non-topmost windows (behind all topmost windows), thus resetting the window's topmost state.
x	The screen position of the left side of the window.
y	The screen position of the top side of the window.
cx	The width of the window.
cy	The height of the window.
uiFlags	Window size and position options. One or more of the following constant values can be OR'ed together:
SWP_DRAWFRAME	Draws a window class's frame around the window.
SWP_HIDEWINDOW	Causes the window to be hidden.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Does not change the window's current position.
SWP_NOSIZE	Does not change the window's current size.
SWP_NOREDRAW	Does not redraw changes. If this flag is set, no part of the window or parent window is redrawn. The application must invalidate the windows' areas and redraw them.
SWP_NOZORDER	Does not change the window ordering.
SWP_SHOWWINDOW	Caused the window to be shown.

40.3 Returns

If the function is successful, the value TRUE is returned. If the function is not successful, the value FALSE is returned.

40.4 Errors

None.

40.5 Cross-References

BeginDeferWindowPos(), DeferWindowPos(), EndDeferWindowPos()

---

41 ShowWindow, IsWindowVisible

41.1 Synopsis

BOOL ShowWindow(HWND hWnd, int nCmdShow);

BOOL IsWindowVisible(HWND hWnd);

41.2 Description

The *ShowWindow()* function sets the given window's show state. *ShowWindow()* must be called only once per program with the *nCmdShow* parameter from the *WinMain()* function. Subsequent calls to *ShowWindow()* must use one of the valid values for the *nCmdShow* parameter, instead of the one given by *nCmdShow* from *WinMain()*.

IsWindowVisible() retrieves the visibility state of the given window. The *hWnd* parameter of the *IsWindowVisible()* function specifies the window to which the visibility state is retrieved.

The *ShowWindow()* function parameters are as follows:

hWnd	This parameter identifies the window for which the show state is set.
nCmdShow	This parameter specifies how the window is to be shown; this parameter can be one of the following values:
SW_HIDE	This value hides the window and activates another window.
SW_MINIMIZE	This value minimizes the window and activates the next top-level window in the Z_order.
SW_RESTORE	This value activates and displays the window; if the window is minimized or maximized, it restores the window to its original size and position.
SW_SHOW	This value activates the window and display it in its current size and position.
SW_SHOWMAXIMIZED	This value activates the window and displays it as a maximized window.
SW_SHOWMINIMIZED	This value activates the window and displays it as a minimized window.
SW_SHOWMINNOACTIVE	This value displays the window as a minimized window; the currently active window remains active.
SW_SHOWNA	This value displays the window in its current state; the currently active window remains active.
SW_SHOWNOACTIVATE	This value displays a window in its most recent size and position; the currently active window remains active.
SW_SHOWNORMAL	This value activates and displays the window; if the window is minimized or maximized, it is restored to its original size and position.

As a result of the *ShowWindow()* function, the following messages are sent to the window's procedure:

WM_SHOWWINDOW, WM_SYSCOMMAND, WM_SIZE, WM_MOVE, WM_ACTIVATE,

WM_NCACTIVATE, WM_WINDOWPOSCHANGING, WM_WINDOWPOSCHANGED

The exact sequence of messages being sent depends on the *nCmdShow* parameter and the current show state of the window.

41.3 Returns

The *IsWindowVisible()* function returns TRUE if the window is visible on the screen (has the WS_VISIBLE style) and the function is successful. Otherwise, it returns FALSE. The *ShowWindow()* function returns TRUE if the window was previously visible on the screen and the function is successful. Otherwise, it returns FALSE.

41.4 Errors

None.

41.5 Cross-References

SetWindowPos(), CreateWindow()

---

42 ArrangeIconicWindows

42.1 Synopsis

UINT ArrangeIconicWindows(HWND hWnd);

42.2 Description

The *ArrangeIconicWindows()* function arranges all iconic (minimized) child windows of the given parent window. The *hWnd* parameter identifies the parent window.

This function can also arrange icons on the desktop if the handle of the desktop window is passed as a parameter.

42.3 Returns

This function returns the height of one row of icons, if it is successful. Otherwise, it returns a zero.

42.4 Errors

None.

42.5 Cross-References

GetDesktopWindow()

---

43 OpenIcon

43.1 Synopsis

BOOL OpenIcon(HWND hWnd);

43.2 Description

The *OpenIcon()* function activates and displays the given minimized window, restoring its original size and position (that is, the size and position the window had before it was minimized). The *hWnd* parameter identifies the window.

Calling the *OpenIcon()* function has the same effect as calling *ShowWindow()* with the SW_SHOWNORMAL flag.

43.3 Returns

This function returns TRUE if successful. Otherwise, it returns FALSE.

43.4 Errors

None.

43.5 Cross-References

ShowWindow(), CloseWindow(), IsIconic()

---

44 BringWindowToTop

44.1 Synopsis

BOOL BringWindowToTop(HWND hWnd);

44.2 Description

The *BringWindowToTop()* function brings the given window to the top of the windows stacking order. In addition, it activates the window, if necessary. Use the *hWnd* parameter to identify the window to bring to the top.

Calling this function has the same effect as calling *SetWindowPos()* with the HWND_TOP value and the SWP_NOMOVE and SWP_NOSIZE flags set.

44.3 Returns

This function returns TRUE if it is successful. Otherwise, it returns FALSE.

44.4 Errors

None.

44.5 Cross-References

SetWindowPos()

45 CloseWindow

45.1 Synopsis

void CloseWindow(HWND hWnd);

45.2 Description

The *CloseWindow()* function minimizes the given window. The *hWnd* parameter identifies the window being minimized.

If the given window is a top-level non-pop-up window, *CloseWindow()* has the same effect as calling *ShowWindow()* with the SW_MINIMIZE flag set. For top-level pop-up windows and child windows, the *CloseWindow()* function has no effect.

45.3 Returns

None.

45.4 Errors

None.

45.5 Cross-References

ShowWindow(), OpenIcon(), IsIconic()

46 IsIconic, IsZoomed

46.1 Synopsis

BOOL IsIconic(HWND hWnd);

BOOL IsZoomed(HWND hWnd);

46.2 Description

The *IsIconic()* function determines whether the given window is minimized. *IsZoomed()* determines whether the given window is maximized.

The *hWnd* parameter specifies the window to query for status.

46.3 Returns

IsIconic() returns TRUE if the window is minimized (has the WS_MINIMIZE style) and the function is successful. Otherwise, it returns FALSE.

IsZoomed() returns TRUE if the window is maximized (has the WS_MAXIMIZE style) and the function is successful. Otherwise, it returns FALSE.

46.4 Errors

None.

46.5 Cross-References

CloseWindow(), OpenIcon(), ShowWindow()

47 ShowOwnedPopups

47.1 Synopsis

void ShowOwnedPopups(HWND hWnd, BOOL fShow);

47.2 Description

The *ShowOwnedPopups()* function shows or hides all pop-up windows owned by the given window. The *hWnd* parameter identifies the owner window. The *fShow* parameter specifies whether pop-up windows are to be shown (TRUE) or hidden (FALSE).

47.3 Returns

None.

47.4 Errors

None.

47.5 Cross-References

IsWindowVisible(), ShowWindow()

48 IsWindow

48.1 Synopsis

BOOL IsWindow(HWND hWnd);

48.2 Description

The *IsWindow()* function determines whether the given window handle identifies the existing window. The *hWnd* parameter specifies the window to query.

48.3 Returns

The *IsWindow()* function returns TRUE if the *hWnd* parameter identifies the existing window and the function is successful. Otherwise, it returns FALSE.

48.4 Errors

None.

48.5 Cross-References

IsChild()

49 IsWindowEnabled, EnableWindow

49.1 Synopsis

BOOL IsWindowEnabled(HWND hWnd);

BOOL EnableWindow(HWND hWnd, BOOL fEnable);

49.2 Description

The *IsWindowEnabled()* function retrieves and the *EnableWindow()* function sets the input status of the given window. When input is disabled, the window ignores input such as mouse clicks and key presses. When input is enabled, the window accepts all input.

The *hWnd* parameter of the *EnableWindow()* function identifies the given window for which the input status is set.

The *fEnable* parameter of *EnableWindow()* specifies whether to enable (TRUE) or disable (FALSE) the window.

If the input state of the window is changing, a WM_ENABLE message is sent to the window with the parameters indicating the new input state. If a window is disabled, all its children, if any, are also disabled, although they do not get the WM_ENABLE message upon input status change.

A window must be enabled before it can be activated. If a child window is disabled, it is ignored by the *WindowFromPoint()* function.

By default, a window is enabled when it is created. To create an initially disabled window, an application can set the WS_DISABLED style bit in the *CreateWindow()* or *CreateWindowEx()* function. An application can set or reset the WS_DISABLED style bit by calling *SetWindowLong()* with the GWL_STYLE offset. The WM_ENABLE message is sent to the window in this case.

49.3 Returns

IsWindowEnabled() returns TRUE if the specified window is enabled and the function is successful. Otherwise, it returns FALSE.

EnableWindow() returns TRUE if the window was previously disabled and the function is successful. Otherwise, it returns FALSE.

49.4 Errors

None.

49.5 Cross-References

CreateWindow(), CreateWindowEx(), SetWindowLong(), WindowFromPoint()

---

50 GetActiveWindow, SetActiveWindow

50.1 Synopsis

HWND GetActiveWindow(void);

HWND SetActiveWindow(HWND hWnd);

50.2 Description

The *GetActiveWindow()* function retrieves and *the SetActiveWindow()* function sets the active window handle. The *hWnd* parameter of the *SetActiveWindow()* function identifies the top-level window to become active. The window, if any, that was previously active, loses its active status, and receives WM_ACTIVATE and WM_NCACTIVATE messages with parameters indicating that the window is being deactivated. The window that becomes active gets the WM_ACTIVATE and WM_NCACTIVATE messages with parameters indicating that the window is being activated. In addition, if the activated window belongs to a different task, the currently active window and the currently active task are deactivated and the new task is activated. This task activation/deactivation process results in sending WM_ACTIVATEAPP messages to the respective window procedures.

50.3 Returns

GetActiveWindow() returns the handle of the active window, if one exists. Otherwise, it returns zero. *SetActiveWindow()* returns the handle of the window that previously was active, if the function is successful. It returns zero, if no window had the active status or if no error has occurred.

50.4 Errors

None.

50.5 Cross-References

None.

---

51 GetCapture, SetCapture, ReleaseCapture

51.1 Synopsis

HWND GetCapture(void);

HWND SetCapture(HWND hWnd);

void ReleaseCapture(void);

51.2 Description

The *GetCapture()* function retrieves and the *SetCapture()* function sets the mouse capture window handle. If the mouse capture is set to a window, all mouse input is directed to that window, regardless of the cursor position. Only one window can have the mouse capture at any given time.

The *ReleaseCapture()* function releases the mouse capture and restores normal input processing.

The *hWnd* parameter of *SetCapture()* identifies the window to receive the capture. The window, if any, that previously had the capture, loses it.

51.3 Returns

GetCapture() returns the handle of the capture window, if one exists. Otherwise, it returns zero. *SetCapture()* returns the handle of the window that previously had the capture, if the function is successful. It returns zero if no window had the capture or if no error has occurred.

ReleaseCapture() does not return a value.

51.4 Errors

None.

51.5 Cross-References

None.

52 GetFocus, SetFocus

52.1 Synopsis

HWND GetFocus(void);

HWND SetFocus(HWND hWnd);

52.2 Description

The *GetFocus()* function retrieves and the *SetFocus()* function sets the input focus window handle. The *hWnd* parameter of the *SetFocus()* function identifies the window to receive the focus. The window, if any, that previously had the focus, loses it, and receives a WM_KILLFOCUS message. The window that receives the focus gets a WM_SETFOCUS message.

SetFocus() also activates either the window that receives the focus or its parent window.

52.3 Returns

GetFocus() returns the handle of the focus window, if one exists. Otherwise, it returns zero. *SetFocus()* returns the handle of the window that previously had the focus, if the function is successful. It returns zero, if no window had the focus or if an error has occurred.

52.4 Errors

None.

52.5 Cross-References

GetActiveWindow(), SetActiveWindow(), GetCapture(), SetCapture()

53 GetSysModalWindow, SetSysModalWindow

53.1 Synopsis

HWND GetSysModalWindow(void);

HWND SetSysModalWindow(HWND hWnd);

53.2 Description

The *GetSysModalWindow()* function retrieves the handle of the system-modal window, if one exists. *SetSysModalWindow()* makes the given window the system-modal window. The *hWnd* parameter of *SetSysModalWindow()* identifies the window to become system-modal.

If another window is made active, it becomes the system-modal window. When the activation is returned to the original window, it becomes the system-modal window again. The system-modal state is ended by destroying the system-modal window.

If the WH_JOURNALRECORD is active when the *SetSysModalWindow()* function is called, the hook function is called with a code parameter of HC_SYSMODALON or HC_SYSMODALOFF.

53.3 Returns

GetSysModalWindow() returns the handle of the system-modal window, if one exists. Otherwise, it returns NULL. *SetSysModalWindow()* returns the handle of the window that was previously the system-modal window, if one existed. It returns NULL if no system-modal window exists or an error has occurred.

53.4 Errors

None.

53.5 Cross-References

SetWindowsHook(), SetWindowHookEx()

---

54 AnyPopup

54.1 Synopsis

BOOL AnyPopup(void);

54.2 Description

The *AnyPopup()* function indicates whether an owned, visible, top-level window exists on the desktop. This function does not detect unowned windows or the windows that do not have the WS_VISIBLE style bit set.

54.3 Returns

This function returns TRUE if such a window exists, even if it is completely obscured by other windows. The function returns FALSE if no such window exists.

54.4 Errors

None.

54.5 Cross-References

ShowOwnedPopups(), EnumWindows(), IsWindowVisible()

---

55 GetLastActivePopup

55.1 Synopsis

HWND GetLastActivePopup(HWND hWndOwner);

55.2 Description

The *GetLastActivePopup()* function retrieves the handle of the most recently active pop-up window owned by the given window. The *hWndOwner* parameter identifies the owner window.

55.3 Returns

This function returns the handle of the most recently active pop-up window when it is successful. Otherwise, it returns zero. The return value is equal to the *hWndOwner* parameter, if any of the following is TRUE:

- the *hWndOwner* window was the most recently active
- the *hWndOwner* window does not own any pop-up windows
- the *hWndOwner* window is not a top-level window or is owned by another window

55.4 Errors

None.

55.5 Cross-References

GetActiveWindow(), ShowOwnedPopups(), AnyPopup()

56 IsChild

56.1 Synopsis

BOOL IsChild(HWND hWndParent, HWND hWnd);

56.2 Description

The *IsChild()* function determines whether the given window is a child window or a descendant window of a given parent window. A child window is a direct descendant of a given parent window, if that parent window is in the chain of parent windows leading from the original top-level window to the child window.

The *hWndParent* parameter specifies the parent window. The *hWnd* parameter specifies the window to be tested.

56.3 Returns

IsChild() returns TRUE if the function is successful. Otherwise, it returns FALSE.

56.4 Errors

None.

56.5 Cross-References

IsWindow()

57 GetParent, SetParent

57.1 Synopsis

HWND GetParent(HWND hWndChild);

HWND SetParent(HWND hWndChild, HWND hWndNewParent);

57.2 Description

The *GetParent()* function retrieves and the *SetParent()* function sets the parent window of the given child window. The *hWndChild* parameter identifies the child window of the associated parent window. The *hWndNewParent* parameter of the *SetParent()* function specifies the new parent window. If this parameter is zero, the desktop window becomes the new parent window. The new parent and the child window must belong to the same application. If the child window is visible, the appropriate window is redrawn.

An application can also set the parent or owner window by calling the *SetWindowWord()* function with GWW_HWNDPARENT offset.

57.3 Returns

GetParent() returns the handle of the parent window for child windows (with the style WS_CHILD) and the handle of the owner window, if any, for pop-up windows (with the style WS_POPUP). It returns zero for overlapping windows or if an error occurs.

SetParent() returns the handle of the previous parent window when the function is successful. Otherwise, it returns zero.

57.4 Errors

None.

57.5 Cross-References

CreateWindow(), SetWindowWord()

58 GetWindow, GetTopWindow, GetNextWindow

58.1 Synopsis

HWND GetWindow(HWND hWnd, UINT uCmd);

HWND GetTopWindow(HWND hWnd);

HWND GetNextWindow(HWND hWnd, UINT uCmd);

58.2 Description

The *GetWindow()* function retrieves the handle of a window that has the specified relationship to the given window. The parameters are as follows:

hWnd	This parameter identifies a window. The window handle retrieved is relative to this window, as specified by the *uCmd* parameter.
uCmd (*GetWindow()*, *GetTopWindow()*)	This parameter specifies the relationship between the given window and the window whose handle is to be retrieved. This parameter can be one of the following values:
GW_CHILD (*GetWindow()* only)	This value retrieves the child window at the top of Z-order, if the give window is a parent window; otherwise returns zero. This function examines only direct child windows (not descendants) of the given window.
GW_HWNDFIRST (*GetWindow()* only)	This value retrieves the sibling window at the top of Z-order.
GW_HWNDLAST (*GetWindow()* only)	This value retrieves the sibling window at the bottom of Z-order.
GW_HWNDNEXT	This value retrieves the sibling below the given window in the Z-order if it is the given window; if the window is the last in the Z-order, returns zero.
GW_HWNDPREV	This value retrieves the sibling above the given window in the Z-order if it is the given window; if the window is the first in the Z-order, returns zero.
GW_OWNER (*GetWindow()* only)	This value retrieves the given window's owner, if any.

Calling *GetTopWindow()* is the same as calling *GetWindow()* with the GW_CHILD parameter.

Calling *GetNextWindow()* is the same as calling the *GetWindow()* function with the GW_HWNDNEXT or GW_HWNDPREV flags.

58.3 Returns

If the function succeeds, the return value is a window handle. If no window exists with the specified relationship to the given window, or in case of error, the return value is zero.

58.4 Errors

None.

58.5 Cross-References

GetActiveWindow()

---

59 MoveWindow

59.1 Synopsis

BOOL MoveWindow(HWND hWnd, int LeftPos, int TopPos, int Width, int Height, BOOL bRepaint);

59.2 Description

The *MoveWindow()* function moves or changes the size of a window. The *hWnd* parameter contains the handle of the window to move or resize. The *LeftPos* parameter contains the new location of the left side of the window. The *TopPos* parameter contains the new location of the top side of the window. The *Width* parameter contains the width of the window. The *Height* parameter contains the height of the window. The *bRepaint* parameter specifies whether the window should be a repainted and sent a WM_PAINT message. The *bRepaint* parameter can be one of the following values:

TRUE	The window is repainted after it is repositioned and receives a WM_PAINT message.
FALSE	The window is not repainted after it is repositioned; any part of the window or its parent window that are uncovered as a result of being repositioned should be redrawn by the application.

A top-level window's position and size are relative to the upper-left corner of the screen. A child window's position and size are relative to its parent's window. The window specified in the *hWnd* parameter is sent a WM_GETMINMAXINFO message as a result of calling the *MoveWindow()* function.

59.3 Returns

If successful, the *MoveWindow()* function returns TRUE. If not successful, the *MoveWindow()* function returns FALSE.

59.4 Errors

None.

59.5 Cross-References

WM_PAINT, WM_GETMINMAXINFO

60 GetDeskTopWindow

60.1 Synopsis

HWND GetDeskTopWindow(void);

60.2 Description

The desktop window is the background window that is covered by all other windows and icons. An application can call the *GetDeskTopWindow()* function to get the window handle of the desktop window.

60.3 Returns

The *GetDeskTopWindow()* function returns the window handle of the desktop window.

60.4 Errors

None.

60.4 Cross-References

None.

61 GetWindowPlacement, SetWindowPlacement

61.1 Synopsis

```
typedef struct tagWINDOWPLACEMENT {
        UINT        length;
        UINT        flags;
        UINT        showCmd;
        POINT       ptMinPosition;
        POINT       ptMaxPosition;
        RECT        rcNormalPosition;
} WINDOWPLACEMENT;
```

BOOL GetWindowPlacement(HWND hWnd, WINDOWPLACEMENT *WndPlacePtr);

BOOL SetWindowPlacement(HWND hWnd, const WINDOWPLACEMENT *WndPlacePtr);

61.2 Description

The *GetWindowPlacement()* function identifies the window's show state (normal, maximized, minimized) and the show state positions and stores the information in a given **WINDOWPLACEMENT** structure. The *hWnd* parameter contains the handle of the window. The *WndPlacePtr* parameter points to a **WINDOWPLACEMENT** structure in which the function will store the window's show state and position information.

The function does not use the **WINDOWPLACEMENT** structure's **flags** element. The function will set the **WINDOWPLACEMENT** structure's **showCmd** element to one of the following values:

SW_SHOWMAXIMIZED	The window is maximized; the position of the window is stored in the **WINDOWPLACEMENT** structure’s **ptMaxPosition** element.
SW_ SHOWMINIMIZED	The window is minimized; the position of the window is stored in the **WINDOWPLACEMENT** structure’s **ptMinPosition** element.
SW_ SHOWNORMAL	The window is not maximized nor minimized; the position of the window is stored in the **WINDOWPLACEMENT** structure's **rcNormalPosition** element.

The *SetWindowPlacement()* function changes the window's show state (normal, maximized, minimized) and the show state positions using the information supplied in the given **WINDOWPLACEMENT** structure. The *hWnd* parameter contains the handle of the window. The *WndPlacePtr* parameter points to a **WINDOWPLACEMENT** structure that contains the window's new show state and show state position information.

61.3 Returns

If successful, both functions return TRUE. If not successful, both functions return FALSE.

61.4 Errors

None.

61.5 Cross-References

WINDOWPLACEMENT

62 AdjustWindowRect, AdjustWindowRectEx

62.1 Synopsis

void AdjustWindowRect(RECT *RectPtr, DWORD dwStyle, BOOL bMenu);

void AdjustWindowRectEx(RECT *RectPtr, DWORD dwStyle, BOOL bMenu, DWORD dwExStyle);

62.2 Description

The *AdjustWindowRect()* and *AdjustWindowRectEx()* functions calculate the size of a window given information about the window's components and styles.

AdjustWindowRect() uses the window's client size, standard styles, and menu information to calculate the window's size. The *RectPtr* parameter is a pointer to a **RECT** structure that has the dimensions of the window's client area. The *dwStyle* parameter specifies the styles of the window. The *bMenu* parameter specifies whether the window has a menu and can be a value of TRUE or FALSE. The *AdjustWindowRect()* function returns the window's calculated size in the **RECT** structure to which the *RectPtr* parameter points.

AdjustWindowRectEx() uses the same window information that is supplied to the *AdjustWindowRect()* function when calculating the window's size. Additionally, the *AdjustWindowRectEx()* function considers the window's extended styles in the calculation. The *AdjustWindowRectEx()* function's first three parameters are the same as the *AdjustWindowRect()* function's parameters. The function's fourth parameter, *dwExStyle*, specifies the window's extended styles.

AdjustWindowRect() and *AdjustWindowRectEx()* do not take window titles or borders into account when computing the window's size. The sizes for these window attributes should be added to the window size returned by the functions. Also, neither function takes extra menu bar rows into account if the window's menu bar wraps to two or more rows.

62.3 Returns

None.

62.4 Errors

None.

62.5 Cross-References

RECT

---

63 GetClientRect, GetWindowRect

63.1 Synopsis

```
typedef struct tagRECT {
        int left;
        int top;
        int right;
        int bottom;
} RECT;
void GetClientRect(HWND hWnd, RECT *RectPtr);
void GetWindowRect(HWND hWnd, RECT *RectPtr);
```

63.2 Description

The *GetClientRect()* function returns the coordinates of window's client area. The *hWnd* parameter contains a handle to the window. The *RectPtr* parameter is a pointer to a **RECT** structure where the *GetClientRect()* function will store the coordinates of window's client area. The **RECT** structure's top and left elements will always be set to zero.

The *GetWindowRect()* function returns a window's screen coordinates relative to the upper-left corner of the display. The *hWnd* parameter contains a handle to the window. The *RectPtr* parameter is a pointer to a **RECT** structure where the *GetWindowRect()* function will store the window's screen coordinates.

63.3 Returns

None.

63.4 Errors

None.

63.5 Cross-References

RECT

---

64 GetWindowText, GetWindowTextLength, SetWindowText

64.1 Synopsis

```
int GetWindowText(HWND hwnd, LPCSTR BufferPtr, int Size);
int GetWindowTextLength(HWND hwnd);
void SetWindowText(HWND hwnd, LPCSTR String);
```

64.2 Description

The *GetWindowText()* function retrieves a window's title or the text within a control and copies it into a buffer. The *hWnd* parameter contains a handle to the window or control. The *BufferPtr* parameter is a pointer to a buffer where the window's title or control's text should be stored. The *Size* parameter specifies the maximum number of bytes that can be copied into the buffer. When this function is used, the system sends a WM_GETTEXT message to the window or control.

The *GetWindowTextLength()* function returns the size, in bytes, of a window's title or the text within a control. The *hWnd* parameter contains a handle to the window or control. When this function is used, the system sends a WM_GETTEXTLENGTH message to the window or control.

The *SetWindowText()* function changes a window's title or sets the text within a control. The *hWnd* parameter contains a handle to the window or control. The *String* parameter is a pointer to a null-terminated string that should be the window's title or the control's text. If the window is a list-box control and has the WS_CAPTION style set, the function will set the caption for the list-box. When this function is used, the system sends a WM_SETTEXT message to the window or control.

64.3 Returns

GetWindowText() returns the size of the text, in bytes, that was copied into the buffer. The return value does not include the text's null-terminating character. Zero is returned if the window handle specified in the *hWnd* parameter is invalid, the window does not have a title, or the control does not contain any text.

GetWindowTextLength() returns the size, in bytes, of a window's title or the text within a control. The return value does not include the text's null-terminating character. Zero is returned if the window handle specified in the *hWnd* parameter is invalid, the window does not have a title, or the control does not contain any text.

SetWindowText() does not return a value.

64.4 Errors

None.

64.5 Cross-References

WM_GETTEXT, WM_GETTEXTLENGTH, WM_SETTEXT

---

65 EnumWindows, EnumWindowsProc

65.1 Synopsis

BOOL EnumWindows(WNDENUMPROC EnumWindowsProc, LPARAM lParam);

BOOL CALLBACK EnumWindowsProc(HWND hWnd, LPARAM lParam);

65.2 Description

The *EnumWindows()* function enumerates all parent windows. The *lParam* parameter contains a user-defined value. The *EnumWindowsProc* parameter is a pointer to an exported, user-defined, callback function that is called each time the *EnumWindows()* function finds a parent window. The *EnumWindows()* function passes the window's handle and the value of the *lParam* parameter to the callback function. The process continues until all of the parent windows are enumerated or until the *EnumWindowsProc()* callback function returns FALSE. The *EnumWindows()* function does not report child windows found during its search.

The *EnumWindowsProc()* function is an exported, user-defined, callback function of type WNDENUMPROC whose address is passed to the *EnumWindows()* function. The *EnumWindows()* function calls the *EnumWindowsProc()* function each time that it finds a parent window. The *hWnd* parameter is a handle to a parent window. The *lParam* parameter is a user-defined value that is passed to the *EnumWindows()* function when it is called.

65.3 Returns

If the *EnumWindows()* function is successful it returns TRUE. If the *EnumWindows()* function is not successful, it returns FALSE.

The *EnumWindowsProc()* function should return TRUE to inform the *EnumWindows()* function to continue enumerating the parent windows. The *EnumWindowsProc()* function should return FALSE to inform the *EnumWindows()* function to stop enumerating the parent windows.

65.4 Errors

None.

65.5 Cross-References

EnumChildWindows()

---

66 EnumChildWindows, EnumChildProc

66.1 Synopsis

BOOL EnumChildWindows(HWND hParentWnd, WNDENUMPROC EnumChildProc, LPARAM lParam);

BOOL CALLBACK EnumChildProc(HWND hWnd, LPARAM lParam);

66.2 Description

The *EnumChildWindows()* function enumerates all of the parent window's child windows. The *hParentWnd* parameter is a handle to a parent window whose child windows will be enumerated. The *lParam* parameter contains a user-defined value. The *EnumChildProc* parameter is a pointer to an exported, user-defined, callback function that is called each time the *EnumChildWindows()* function finds a child window. The *EnumChildWindows()* function passes the child window's handle and the value of the *lParam* parameter to the callback function. The process continues until all of the child windows are enumerated or until the *EnumChildProc()* callback function returns FALSE.

The *EnumChildProc()* function is an exported, user-defined, callback function of type WNDENUMPROC whose address is passed to the *EnumChildWindows()* function. The *EnumChildWindows()* function calls the *EnumChildProc()* function each time that it finds a child window. The *hWnd* parameter is a handle to a child window. The *lParam* parameter is a user-defined value that is passed to the *EnumChildWindows()* function when it is called.

66.3 Returns

If the *EnumChildWindows()* function is successful, it returns TRUE. If the *EnumChildWindows()* function is not successful, it returns FALSE.

The *EnumChildProc()* function should return TRUE to inform the *EnumChildWindows()* function to continue enumerating the child windows. The *EnumChildProc()* function should return FALSE to inform the *EnumChildWindows()* function to stop enumerating the child windows.

66.4 Errors

None.

66.5 Cross-References

EnumWindows()

---

67 FindWindow

67.1 Synopsis

HWND FindWindow(LPCSTR ClassName, LPCSTR WindowTitle);

67.2 Description

The FindWindow() function searches for a window using the following criteria:

- window's class name
- window's title

The *ClassName* parameter is a pointer to a null-terminated string that contains the window class for which to search. If the value of the *ClassName* parameter is NULL, all window class names satisfy the search criteria.

The *WindowTitle* parameter is a pointer to a null-terminated string that contains the window title for which to search. If the value of the *WindowTitle* parameter is NULL, all window titles satisfy the search criteria.

67.3 Returns

If the *FindWindow()* function is successful it returns the handle of the first window that matches the search criteria. The *FindWindow()* function returns NULL if it cannot find a window that matches the search criteria.

67.4 Errors

None.

67.5 Cross-References

None.

---

68 AppendMenu, InsertMenu, ModifyMenu

68.1 Synopsis

BOOL AppendMenu(HMENU hMenu, UINT uiFlags, UINT uiIDNewItem, LPCSTR lpNewItem);

BOOL InsertMenu(HMENU hMenu, UINT uiPosition, UINT uiFlags, UINT uiIDNewItem,

LPCSTR lpNewItem);

BOOL ModifyMenu(HMENU hMenu, UINT uiPosition, UINT uiFlags, UINT uiIDNewItem,

LPCSTR lpNewItem);

68.2 Description

The *AppendMenu()*, *InsertMenu()*, and *ModifyMenu()* functions operate on an existing menu specified in the hMenu parameter.

AppendMenu() appends a new item to the end of the menu. The *InsertMenu()* function inserts a new item before the item specified in the *uiPosition* parameter as determined by *uiFlags*.

For *InsertMenu()*, if the MF_BYPOSITION bit is set and *uiPosition* is (-1), the new item appends to the end of the menu. The *uiIDNewItem* parameter is the identifier value for the new item. If the MF_POPUP bit in *uiFlags* is set, this parameter specifies the handle of a pop-up menu. The *lpNewItem* parameter specifies the contents of the new item. If the MF_BITMAP bit is set in *uiFlags*, *lpNewItem* contains the bitmap handle. If the MF_OWNERDRAW flag is specified, it contains the application-defined value. Otherwise when the MF_STRING bit is set, *lpNewItem* is a pointer to a null-terminated string.

ModifyMenu() changes the existing menu item specified in *uiPosition*. The *uiFlags* parameter determines the interpretation of the *uiPosition* value in *InsertMenu()* and *ModifyMenu()*. If the MF_BYPOSITION bit in *uiFlags* is set, *uiPosition* specifies the zero-based position of the menu item in the menu. Otherwise when the MF_BYCOMMAND bit is set, *uiPosition* contains the identifier of the menu item, and the item can be contained either in the menu specified in the *hMenu* parameter or in one of its sub-menus.

In addition to the flags described above, *uiFlags* can be a combination of the following values:

MF_SEPARATOR,

MF_ENABLED, MF_GRAYED, MF_DISABLED,

MF_UNCHECKED, MF_CHECKED,

MF_MENUBREAK, MF_MENUBARBREAK

68.3 Returns

These functions return TRUE, if they are successful. Otherwise, they return FALSE.

68.4 Errors

None.

68.5 Cross-References

None.

69 RemoveMenu, DeleteMenu, DestroyMenu

69.1 Synopsis

BOOL RemoveMenu(HMENU hMenu, UINT idItem, UINT uiFlags);

BOOL DeleteMenu(HMENU hMenu, UINT idItem, UINT uiFlags);

BOOL DestroyMenu(HMENU hMenu);

69.2 Description

The *RemoveMenu()* function removes a pop-up menu item from a menu, but does not destroy the pop-up menu handle. *DeleteMenu()* removes an item from a menu. If the item is a pop-up menu, the handle of the pop-up menu is destroyed and all the memory associated with it is freed. *DestroyMenu()* destroys the menu handle specified in the *hMenu* parameter and frees all the memory it uses.

The *hMenu* parameter is a handle of the menu that contains the given item. The *idItem* parameter defines the menu item to be removed. The *uiFlags* value determines whether the *idItem* parameter contains the position of the item in the menu (the MF_BYPOSITION bit in *uiFlags* is set) or the menu-item identifier (the MF_BYCOMMAND bit in *uiFlags* is set).

69.3 Returns

These functions return TRUE if they are successful. Otherwise, they return FALSE.

69.4 Errors

None.

69.5 Cross-References

None.

70 CreateMenu, CreatePopupMenu

70.1 Synopsis

HMENU CreateMenu(void);

HMENU CreatePopupMenu(void);

70.2 Description

The *CreateMenu()* function creates a new menu. *CreatePopupMenu()* creates a new pop-up menu. The newly created menus are empty, and they can now be filled by calls to *AppendMenu()* or *InsertMenu()*.

DestroyMenu() is called to dispose of the menu created by these functions.

70.3 Returns

These functions return the handle of the created menu, if they are successful. Otherwise, they return zero.

70.4 Errors

None.

70.5 Cross-References

None.

71 GetMenu

71.1 Synopsis

HMENU GetMenu(HWND hWnd);

71.2 Description

The *GetMenu()* function obtains the handle of the menu associated with the window specified in the *hWnd* parameter. This call is usually used only for a top-level window.

71.3 Returns

This function returns the handle of the window menu, if it is successful. Otherwise, it returns zero.

71.4 Errors

None.

71.5 Cross-References

SetMenu()

---

72 DrawMenuBar

72.1 Synopsis

void DrawMenuBar(HWND hWnd);

72.2 Description

The *DrawMenuBar()* function redraws the menu bar of the window specified in the *hWnd* parameter.

72.3 Returns

None.

72.4 Errors

None.

72.5 Cross-References

None.

---

73 GetSystemMenu

73.1 Synopsis

HMENU GetSystemMenu(HWND hWnd, BOOL bRevert);

73.2 Description

The *GetSystemMenu()* function retrieves the handle of the system menu for the window specified in the *hWnd* parameter.

The *bRevert* parameter determines the action to be performed. If *bRevert* is FALSE, the function obtains the handle of the copy of the system menu currently associated with the given window. When the window is created, it receives a copy of the standard system menu that can later be modified by the application. The menu retrieved in this case can be different from the standard system menu, if the application has already modified it.

If *bRevert* is TRUE, the window receives a new copy of the standard system menu and the current one is destroyed.

73.3 Returns

If the *bRevert* parameter is FALSE, the function returns the handle of the current system menu. Otherwise, the return value is undefined.

73.4 Errors

None.

73.5 Cross-References

AppendMenu(), InsertMenu(), ModifyMenu()

---

74 CheckMenuItem, EnableMenuItem, HiliteMenuItem

74.1 Synopsis

BOOL CheckMenuItem(HMENU hMenu, UINT uiIDItem, UINT uiFlags);

BOOL EnableMenuItem(HMENU hMenu, UINT uiIDItem, UINT uiFlags);

BOOL HiliteMenuItem(HMENU hMenu, UINT uiIDItem, UINT uiFlags);

74.2 Description

The *CheckMenuItem()*, *EnableMenuItem()*, and *HiliteMenuItem()* functions change the state of a menu item from the menu specified in the *hMenu* parameter. The *uiIDItem* parameter defines the item that is affected. The *uiFlags* parameter determines how to treat the *uiIDItem* parameter (MF_BYCOMMAND or MF_BYPOSITION) and how the state of the item is modified.

If the MF_BYPOSITION bit in the *uiFlags* parameter is set, the *uiIDItem* parameter specifies the zero-based index of the item in the menu. Otherwise, the *uiIDItem* value is the menu item identifier. If the menu item is searched by its identifier, it can belong to either the menu specified by *hMenu* or to any one of its submenus. The exception is the *HiliteMenuItem()* function that can operate only on the top-level (menubar) menu items.

CheckMenuItem() checks for MF_CHECKED and MF_UNCHECKED flags in *uiFlags*. If MF_CHECKED is set, the checkmark bitmap is placed next to the menu item. Otherwise, the checkmark is cleared.

EnableMenuItem() checks the MF_ENABLED, MF_DISABLED, and MF_GRAYED flags. If the menu item is disabled or grayed, the function does not respond to user input selection.

HiliteMenuItem() checks the MF_HILITE and MF_UNHILITE flags in *uiFlags*. If the MF_HILITE flag is set, the item is highlighted (selected). Otherwise, the highlight is removed. The MF_HILITE and MF_UNHILITE flags cannot be used with the *ModifyMenu()* function.

74.3 Returns

These functions return TRUE, if they are successful. Otherwise, they return FALSE. The functions fail if *hMenu* is invalid or if the requested menu item cannot be located.

74.4 Errors

None.

74.5 Cross-References

GetMenuState(), ModifyMenu()

---

75 GetMenuItemID, GetSubMenu

75.1 Synopsis

UINT GetMenuItemID(HMENU hMenu, int nPos);

HMENU GetSubMenu(HMENU hMenu, int nPos);

75.2 Description

The *GetMenuItemID()* function obtains the menu item identifier of an item from a menu specified in the *hMenu* parameter. This function cannot be used if the item is a pop-up menu. *GetSubMenu()* retrieves a handle of a pop-up menu that is an item on a menu specified by *hMenu*.

The *nPos* parameter provides the zero-based position of the item in the menu.

75.3 Returns

GetMenuItemID() returns the identifier of the given item. It returns zero if the function is unsuccessful or if the item is a separator (has the MF_SEPARATOR flag set). If the menu item is a pop-up menu, the function returns -1. *GetSubMenu()* returns a pop-up menu handle. These functions fail if the *hMenu* parameter is invalid or if the menu item cannot be located.

75.4 Errors

None.

75.5 Cross-References

GetMenuString(), GetMenuState()

76 GetMenuState, GetMenuString

76.1 Synopsis

UINT GetMenuState(HMENU hMenu, UINT uiIDItem, UINT uiFlags);

int GetMenuString(HMENU hMenu, UINT uiIDItem, LPSTR lpString, int nMaxCount, UINT uiFlags);

76.2 Description

The *GetMenuState()* function returns the flags for a menu item on the menu specified in the *hMenu* parameter. *GetMenuString()* retrieves the contents string of a menu item from the menu specified in the *hMenu* parameter and copies it into the buffer passed to it in the *lpString* parameter. The *nMaxCount* parameter indicates the maximum length of the string that the buffer can hold. The menu string copied into the buffer is truncated, if necessary, and the string is null-terminated.

The *uiIDItem* parameter defines the menu item as determined by the *uiFlags* value. If the *uiFlags* value has the MF_BYPOSITION bit set, the *uiIDItem* parameter specifies a zero-based position of the item in the menu. If *uiFlags* equals MF_BYCOMMAND, the *uiIDItem* parameter is a menu-item identifier. In this case, the menu item can belong to either the menu specified in *hMenu* or to one of its sub-menus.

76.3 Returns

GetMenuState() returns the state flags of the item, if the function is successful. If unsuccessful, it returns a -1. *GetMenuString()* returns the length of the menu item string not including the terminating null character, if the function is successful. Otherwise, it returns zero. Both functions fail if the specified *hMenu* is invalid or if the requested menu item cannot be located.

76.4 Errors

None.

76.5 Cross-References

GetMenuItemID(), CheckMenuItem(), EnableMenuItem(), HiliteMenuItem()

77 GetMenuItemCount

77.1 Synopsis

int GetMenuItemCount(HMENU hMenu);

77.2 Description

The *GetMenuItemCount()* function retrieves the number of items in a menu specified in *hMenu* parameter.

77.3 Returns

The function returns the item count, if it is successful. Otherwise, it returns -1. The function fails if the specified *hMenu* is invalid.

77.4 Errors

None.

77.5 Cross-References

None.

78 GetMenuCheckMarkDimensions

78.1 Synopsis

DWORD GetMenuCheckMarkDimensions(void);

78.2 Description

The *GetMenuCheckMarkDimensions()* function retrieves the size of the default check mark bitmap. The check mark is displayed next to the menu item that is checked. The MF_CHECKED bit is set either by a call to *CheckMenuItem()* or by *ModifyMenu()*.

Before an application calls the *SetMenuItemBitmaps()* function to replace the default check mark bitmap with its own check mark bitmap, it should first call this function to determine the correct size of the custom bitmaps to be installed.

78.3 Returns

The function returns a doubleword containing the width in pixels of the default bitmap in the low-order word and the height in pixels in the high-order word.

78.4 Errors

None.

78.5 Cross-References

SetMenuItemBitmaps()

---

79 SetMenuItemBitmaps

79.1 Synopsis

BOOL SetMenuItemBitmaps(HMENU hMenu, UINT uiItem, UINT uiFlags,

HBITMAP hBitmapUnchecked, HBITMAP hBitmapChecked);

79.2 Description

The *SetMenuItemBitmaps()* function sets the custom check mark bitmaps for a menu item from the menu specified in the *hMenu* parameter. These bitmaps are displayed next to the given menu item depending on its state (checked or unchecked).

The *uiItem* parameter specifies the menu item to be affected. The *uiFlags* parameter determines how *uiItem* identifies the item. If the *uiFlags* parameter is set to MF_BYPOSITION, *uiItem* is the zero-based position of the item in *hMenu*. Otherwise if *uiFlags* is set to MF_BYCOMMAND, it is a menu item identifier of an item in either *hMenu* or in one of its sub-menus.

The *hBitmapUnchecked* parameter is a handle of a bitmap to be displayed when the item is unchecked. The *hBitmapChecked* parameter is a handle of a bitmap to be placed next to the item when it is checked.

GetMenuCheckMarkDimensions() is called to determine the size of the default check mark bitmap to be replaced.

If either one of the bitmap handles is zero, nothing is displayed next to the item in the appropriate state. If both handles are zero, the default check mark bitmap is displayed for the checked state. Nothing is drawn for the unchecked state.

79.3 Returns

This function returns TRUE, if it is successful. Otherwise, it returns FALSE. This function fails if the specified hMenu is invalid or if the menu item cannot be located.

79.4 Errors

None.

79.5 Cross-References

GetMenuCheckMarkDimensions()

---

80 TrackPopupMenu

80.1 Synopsis

BOOL TrackPopupMenu(HMENU hMenu, UINT uiFlags, int x, int y, int nReserved, HWND hWnd,

LPRECT lpRect);

80.2 Description

The *TrackPopupMenu()* function displays the pop-up menu handle of the menu specified in the *hMenu* parameter and tracks the selection events resulting from mouse or keyboard input to the specified menu and its sub-menus.

The *x* and *y* parameters specify the screen coordinates of the pop-up menu. The menu can be displayed anywhere on the screen. The *uiFlags* parameter determines the position of the pop-up menu relative to the *x* and *y* coordinates and relative to the mouse-button flags.

The position flag can be one of the following:

TPM_CENTERALIGN	The menu is centered horizontally relative to the *x* position.
TPM_LEFTALIGN	The menu is left-aligned with the *x* position.
TPM_RIGHTALIGN	The pop-up menu is right-aligned with the *x* position.

The mouse-button flag can be either TPM_LEFTBUTTON or TPM_RIGHTBUTTON. This flag determines which mouse button events are tracked by the pop-up menu. The *nReserved* parameter must be set to zero.

The *hWnd* parameter specifies a handle of the window that owns the pop-up menu. This window receives notification messages (WM_MENUSELECT, WM_INITMENUPOPUP) from the menu, as well as the WM_COMMAND message when the function returns.

The *lpRect* parameter points to a **RECT** structure that specifies the screen coordinates of a rectangle in which the mouse button clicks will prevent the pop-up window from being dismissed. If this parameter is NULL, the pop-up menu is dismissed on a button-click outside the pop-up menu.

80.3 Returns

The function returns TRUE, if it is successful. Otherwise, it returns FALSE. This function fails if an invalid menu handle, insufficient memory, invalid screen coordinates, or an invalid window handle is used.

80.4 Errors

None.

80.5 Cross-References

None.

81 SetMenu

81.1 Synopsis

BOOL SetMenu(HWND hWnd, HMENU hMenu);

81.2 Description

The *SetMenu()* function associates a menu with a window specified in the *hWnd* parameter. The *hMenu* parameter is a handle of the menu. If it is zero, the current window's menu will be removed.

The window's menu bar is redrawn as a result of this call.

81.3 Returns

This function returns TRUE if it is successful. Otherwise, it returns FALSE.

81.4 Errors

None.

81.5 Cross-References

GetMenu()

82 IsMenu

82.1 Synopsis

BOOL IsMenu(HMENU MenuHndl);

82.2 Description

The *IsMenu()* function identifies whether the given handle is a menu handle. The *MenuHndl* parameter specifies the handle to be verified. This function is normally used to determine if the given menu handle is *not* a menu handle.

82.3 Returns

This function returns FALSE if the menu handle is *not* a menu handle. If it returns TRUE, it guarantees that the given handle is a menu handle. It is the application programmer's responsibility to conduct further tests to determine the type of handle.

82.4 Errors

None.

82.5 Cross-References

CreateMenu(), CreatePopupMenu(), DestroyMenu(), GetMenu()

---

83 GetScrollPos, SetScrollPos

83.1 Synopsis

int GetScrollPos(HWND hWnd, int flags);

int SetScrollPos(HWND hWnd, int flags, int nPos, BOOL fRepaint);

83.2 Description

The *GetScrollPos()* function returns the current position of the specified scrollbar. The *hWnd* parameter contains the handle of the window possessing the scrollbar or the handle of the scrollbar itself, if the scrollbar is a control. The *flags* parameter indicates which scrollbar is to be queried. SB_HORZ queries the horizontal scrollbar in the indicated window. SB_VERT queries the vertical scrollbar. The SB_CTL value queries the position of the scrollbar control whose handle is passed in the *hWnd* parameter.

The *SetScrollPos()* function sets the current position of a scrollbar, possibly redrawing it in the process. The *hWnd* and *flags* parameters are identical to those in the *GetScrollPos()* call. The *nPos* parameter indicates the new position of the scrollbar. The *fRepaint* parameter is a flag indicating whether the scrollbar should be redrawn.

83.3 Returns

GetScrollPos(), if it is successful, returns the position of the indicated scrollbar. Otherwise, it returns zero.

SetScrollPos() returns the previous position of the indicated scrollbar, if it is successful. Otherwise, it returns zero.

If the window handle in the *hwnd* parameter does not contain the indicated horizontal or vertical scrollbar or is not a scrollbar control, both the *GetScrollPos()* and the *SetScrollPos()* functions return zero.

83.4 Errors

None.

83.5 Cross-References

GetScrollRange(), SetScrollRange()

---

84 GetScrollRange, SetScrollRange

84.1 Synopsis

void GetScrollRange(HWND hWnd, int flags, int *pnMinPos, int *pnMaxPos);

void SetScrollRange(HWND hWnd, int flags, int nMinPos, int nMaxPos, BOOL fRedraw);

84.2 Description

The *GetScrollRange()* function returns the range of the specified scrollbar. The *hWnd* parameter is the handle of the window containing the scrollbar to be queried or is the handle of a scrollbar control. The *flags* parameter indicates which scrollbar is to be queried. Use SB_HORZ to query the horizontal scrollbar of the specified window, use SB_VERT to query the vertical scrollbar, or use SB_CTL to query the scrollbar control whose window handle is passed as the *hWnd* parameter. The *pnMinPos* and *pnMaxPos* parameters are pointers to integers that will receive the minimum and maximum positions of the scrollbar, respectively. If the scrollbar's range has not been set by using the *SetScrollRange()* function, the range will be from 0 through 100.

SetScrollRange() sets the range of the specified scrollbar. The *hWnd* and *flags* parameters specify which scrollbar's range is to be set, as in the *GetScrollRange()* function. The *nMinPos* and *nMaxPos* parameters indicate the new range of the scrollbar. If both the *nMinPos* and *nMaxPos* parameters are zero, the scrollbar will be hidden. If either parameter is non-zero, the scrollbar will be shown. The *fRedraw* parameter indicates whether the scrollbar should be redrawn. If *fRedraw* is TRUE, the scrollbar control will be immediately redrawn.

84.3 Returns

None.

84.4 Errors

None.

84.5 Cross-References

GetScrollPos(), SetScrollPos(), ShowScrollBar()

---

85 ShowScrollBar

85.1 Synopsis

void ShowScrollBar(HWND hWnd, int flags, BOOL fShow);

85.2 Description

ShowScrollBar() shows or hides a scrollbar. The *hWnd* parameter contains the handle of the window possessing the scrollbar or the handle of an independent scrollbar control. The *flags* parameter indicates which scrollbar is being shown or hidden. If the flag is SB_HORZ, the horizontal scrollbar of the specified window is affected. If the flag is SB_VERT, the vertical scrollbar of the specified window is affected. SB_BOTH shows or hides both the horizontal and vertical scrollbars. SB_CTL is used to show or hide a scrollbar control whose window handle is passed as the *hWnd* parameter. The *fShow* parameter indicates whether the specified scrollbar is to be shown or hidden. If it is TRUE, the scrollbar is shown. If it is FALSE, the scrollbar is hidden.

85.3 Returns

None.

85.4 Errors

None.

85.5 Cross-References

GetScrollRange(), SetScrollRange()

---

86 ScrollWindow

86.1 Synopsis

void ScrollWindow(HWND hWnd, int Horz, int Vert, const RECT *ScrollRectPtr,

const RECT *ClipRectPtr);

86.2 Description

The *ScrollWindow()* function scrolls what is shown in a window's client area. The *hWnd* parameter contains the handle of the window. The *Horz* parameter contains the number of device units to scroll the client area horizontally. A positive value for *Horz* scrolls the window to the right, a negative value scrolls it to the left. The *Vert* parameter contains the number of device units to scroll the client area vertically. A positive value for *Vert* scrolls the down, a negative value scrolls it up. The *ScrollRectPtr* parameter points to a RECT structure that identifies the part of the client area to be scrolled. If the *ScrollRectPtr* parameter is NULL, the entire client area will be scrolled. The *ClipRectPtr* parameter points to a **RECT** structure that the clipping rectangle is to scroll. Only the client area defined by this rectangle is scrolled. The client area outside this rectangle is not scrolled even if the area is inside the rectangle specified in the *ScrollRectPtr*. If the *ClipRectPtr* parameter is NULL, no clipping is performed on the scroll rectangle specified in the *ScrollRectPtr*.

If the window being scrolled contains the caret or cursor, it will automatically be hidden before the scroll and restored after the scroll. If the rectangle for the caret or cursor intersects the scrolled area, its position is moved accordingly.

The screen area uncovered by the scroll is not repainted and is combined into the window's update region. A WM_PAINT message is sent to the window to tell it that it needs to repaint the uncovered areas. To force an immediate repainting of the window, call the *UpdateWindow()* function after calling the *ScrollWindow()* function.

If the value of the *ScrollRectPtr* parameter is NULL, the window's child windows and invalid paint areas are offset by the movement of the scroll operation.

If the value of the *ScrollRectPtr* parameter is not NULL, the positions of child windows are not changed and invalid paint areas in the window are not offset by the movement of the scroll operation. When *ScrollRectPtr* is not NULL, to prevent updating problems, call the *UpdateWindow()* function to repaint the window before calling the *ScrollWindow()* function.

86.3 Returns

None.

86.4 Errors

None.

86.5 Cross-References

RECT, *UpdateWindow()*

87 EnableScrollBar

87.1 Synopsis

BOOL EnableScrollBar(HWND hWnd, int SBFlagType,UINT SBArrowFlags);

87.2 Description

The *EnableScrollBar()* function enables or disables one or both of the arrows of the scroll bar. The *hWnd* parameter specifies the handle of the window or scroll bar, depending on the value of the *SBFlagType* parameter. This parameter can have any one of the following values:

SB_BOTH	This value enables or disables the arrows of the horizontal and vertical scroll bars associated with the given window; the *hWnd* parameter identifies the window.
SB_CTRL	This value identifies the scroll bar as a scroll bar control; the *hWnd* parameter must identify a scroll bar control.
SB_HORZ	This value enables or disables the arrows of the horizontal scroll bar associated with the given window; the *hWnd* parameter identifies the window.
SB_VERT	This value enables or disables the arrows of the vertical scroll bar associated with the given window; the *hWnd* parameter identifies the window.

The *SBArrowFlags* parameter identifies whether the arrows in the scroll bar are enabled or not, and also which arrows are enabled or disabled. This parameter can have one of the following values.

ESB_ENABLE_BOTH	This value enables both arrows of a scroll bar.
ESB_DISABLE_LTUP	This value disables the left arrow of a horizontal scroll bar, or the up arrow of a vertical scroll bar.
ESB_DISABLE_RTDN	This value disables the right arrow of a horizontal scroll bar, or the down arrow of a vertical scroll bar.
ESB_DISABLE_BOTH	This value disables both arrows of a scroll bar.

87.3 Returns

The function returns TRUE if the arrows were enabled or disabled. It returns FALSE to indicate that the arrows are already in the state in which they were required to be set or that an error occurred.

87.4 Errors

None.

87.5 Cross-References

ShowScrollBar()

---

88 ScrollDC

88.1 Synopsis

BOOL ScrollDC(HDC hdc, int dx, int dy, const RECT *lprcScroll, const RECT *lprcClip,

HRGN hrgnUpdate, RECT *lprcUpdate);

88.2 Description

The *ScrollDC()* function scrolls the given device-context horizontally or vertically. The *lprcScroll* parameter points to the **RECT** structure with the coordinates of the scrolling rectangle. The *lprcClip* parameter points to the **RECT** with the coordinates of the clipping rectangle. Scrolling occurs only in the smallest of either two rectangles. If the *hrgnUpdate* parameter is not NULL, it defines the region, uncovered by the scrolling, otherwise, the update region is not computed. If the *lprcUpdate* parameter is not NULL, it receives the client coordinates of the largest rectangular area that needs repainting. Otherwise, the update rectangle is not computed.

88.3 Returns

The function returns TRUE if it is successful. Otherwise, it returns FALSE.

88.4 Errors

None.

88.5 Cross-References

InvalidateRgn(), ScrollWindow(), ScrollWindowEx()

---

89 ScrollWindowEx

89.1 Synopsis

int ScrollWindowEx(HWND hWnd, int Horz, int Vert, const RECT *ScrollRectPtr,

const RECT *ClipRectPtr, HRGN hUpdateRgn, RECT *UpdateRectPtr, UINT ScrollFlag);

89.2 Description

The *ScrollWindowEx()* function scrolls what is shown in a window's client area and is an extension of the *ScrollWindow()* function. The *hWnd* parameter contains the handle of the window. The *Horz* parameter contains the number of device units to scroll the client area horizontally. A positive value for *Horz* scrolls the window to the right; a negative value scrolls it to the left. The *Vert* parameter contains the number of device units to scroll the client area vertically. A positive value for *Vert* scrolls the down; a negative value scrolls it up. The *ScrollRectPtr* parameter points to a **RECT** structure that identifies the part of the client area to be scrolled. If the *ScrollRectPtr* parameter is NULL, the entire client area is scrolled. The *ClipRectPtr* parameter points to a **RECT** structure that the clipping rectangle should scroll. Only the client area defined by this rectangle is scrolled. The client area outside this rectangle is not scrolled even if the area is inside of the rectangle specified in the *ScrollRectPtr*.

If the *ClipRectPtr* parameter is NULL, no clipping is performed on the scroll rectangle specified in the *ScrollRectPtr*. The *hUpdateRgn* parameter is a handle to a region in which the *ScrollWindowEx()* function will save the region area that was invalidated by the scrolling action. If the value of the *hUpdateRgn* parameter is NULL, the function will not attempt to store the region of the client area that was invalidated by the scrolling action. The *UpdateRectPtr* parameter points to a **RECT** structure in which the *ScrollWindowEx()* function will save the rectangular area of the client area that was invalidated by the scrolling action. If the value of the *UpdateRectPtr* parameter is NULL, the function will not attempt to store the invalidated region's rectangular area.

The *ScrollFlag* parameter specifies other scrolling information and can be one or more of the following flags combined together using the OR (|) operation:

SW_ERASE	This flag causes a WM_ERASEBKGND message to be sent to the window; if this flag is used with the SW_INVALIDATE flag, the region invalidated by the scrolling action will be erased.
SW_INVALIDATE	After the scrolling action is performed, the *ScrollWindowEx()* function will invalidate the region identified by the *hUpdateRgn* parameter.
SW_SCROLLCHILDREN	The *ScrollWindowEx()* function will scroll all of the child windows that intersect with the rectangle given in the *ScrollRectPtr* parameter. The child windows will be scrolled by the number of pixels in the parameters *Horz* and *Vert*, and a WM_MOVE message will be sent to all of the child windows that intersect with the rectangle given in the *ScrollRectPtr* parameter. If the caret rectangle intersects the rectangle that is scrolled, the cursor will be repositioned. The system will not properly update the screen when a part of a child window is scrolled; the area of the scrolled child window that is outside of the *ScrollRectPtr* rectangle will not be erased and will not be redrawn properly in its new destination. The application can use the *DeferWindowPos()* function to move child windows that are not completely within the scrolled area.

If the window specified in the *hWnd* parameter has the WS_CLIPCHILDREN style set, the areas stored in the *hUpdateRgn* and *UpdateRectPtr* parameters will be the total area, including any areas in child windows, that should be updated.

89.3 Returns

If the function is successful, it returns one of the following values:

SIMPLEREGION	This value specifies a rectangular invalidated region.
COMPLEXREGION	This value specifies a non-rectangular, invalidated region of overlapping rectangles.
NULLREGION	This value specifies no invalidated region.

If the function is not successful, it returns ERROR.

89.4 Errors

None.

89.5 Cross-References

RECT, *ScrollWindow(), DeferWindowPos()*

90 CreateCaret, DestroyCaret

90.1 Synopsis

void CreateCaret(HWND hWnd, HBITMAP hBitmap, int nWidth, int nHeight);

void DestroyCaret(void);

90.2 Description

The *CreateCaret()* function creates a new image for the system caret. The *hWnd* parameter contains the window handle of the window that owns the caret. The *hBitmap* parameter holds the handle of a bitmap that contains the image for the caret. It can also be NULL, in which case a solid black caret will be used, or the value 1, cast to an HBITMAP, will cause a solid gray caret to be used. If the *hBitmap* parameter is either NULL or 1, the *nWidth* and *nHeight* parameters indicate the width and height of the caret image. If the *nWidth* parameter is zero, the system window-border width is used as the caret width. If the *nHeight* parameter is zero, the window-border height is used as the caret height.

The *DestroyCaret()* function destroys the current caret image. It uses no parameters.

CreateCaret() replaces the current caret image with a new image. If the current window does not have the input focus, the *CreateCaret()* and *DestroyCaret()* functions should not be called. The *ShowCaret()* function should be called to make the new caret visible.

90.3 Returns

None.

90.4 Errors

None.

90.5 Cross-References

ShowCaret()

---

91 GetCaretBlinkTime, SetCaretBlinkTime

91.1 Synopsis

UINT GetCaretBlinkTime(void);

void SetCaretBlinkTime(UINT msBlink);

91.2 Description

The *GetCaretBlinkTime()* function returns the length of time required for the caret blink. It has no parameters. *SetCaretBlinkTime()* sets the length of time required for the caret blink. The *msBlink* parameter indicates the number of milliseconds for the blink. Every *msBlink* milliseconds the caret turns on or off. One complete caret blink requires twice the length of time specified in the *msBlink* parameter.

An application should only set the caret blink time if it has the input focus and owns the caret. It should reset the blink time when it loses the input focus or destroys the caret.

91.3 Returns

GetCaretBlinkTime() returns the current caret blink time, in milliseconds. *SetCaretBlinkTime()* does not return a value.

91.4 Errors

None.

91.5 Cross-References

None.

---

92 GetCaretPos, SetCaretPos

92.1 Synopsis

```
typedef struct tagPOINT {
        int x;
        int y;
} POINT;
```

void GetCaretPos(POINT *pPoint);

void SetCaretPos(int x, int y);

92.2 Description

The *GetCaretPos()* function queries the current position of the caret. The *pPoint* parameter contains a pointer to a **POINT** structure. The caret position returns using the **x** and **y** members of this structure.

SetCaretPos() sets the current caret position. The *x* and *y* parameters contain the new coordinates of the caret, relative to the origin of the window that currently owns it.

92.3 Returns

Neither function returns a value. *GetCaretPos()* returns the current caret position in the structure pointed to by the *pPoint* parameter.

92.4 Errors

None.

92.5 Cross-References

None.

---

93 HideCaret, ShowCaret

93.1 Synopsis

void HideCaret(HWND hWnd);

void ShowCaret(HWND hWnd);

93.2 Description

The *HideCaret()* function removes the caret from the screen if it has been called at least as many times as the *ShowCaret()* function. The *hWnd* parameter contains the handle of the window that currently owns the caret.

ShowCaret() displays the caret on the screen if it has been called more times than the *HideCaret()* function. The *hWnd* parameter contains the handle of the window that currently owns the caret.

If the *hwnd* parameter to either function is NULL, the caret is only shown or hidden if it is owned by a window in the current task.

93.3 Returns

None.

93.4 Errors

None.

93.5 Cross-References

None.

---

94 CreateCursor, DestroyCursor

94.1 Synopsis

HCURSOR CreateCursor(HINSTANCE hInstance, int xHotSpot, int yHotSpot, int nWidth, int nHeight,

    const void *pANDplane, const void *pXORplane);

BOOL DestroyCursor(HCURSOR hCursor);

94.2 Description

The *CreateCursor()* function creates a new cursor. The *hInstance* parameter contains the instance handle of the application creating the cursor. The *xHotSpot* and *yHotSpot* parameters indicate the coordinates of the cursor's hotspot. The hotspot is the location on the cursor bitmap where the "point" of the cursor is located (for example, the point of an arrow or the center of a cross). The *nWidth* and *nHeight* parameters indicate the width and height of the new cursor, in pixels.

The *pANDplane* and *pXORplane* parameters contain arrays of bytes containing the bit values of masks for the cursor. In the AND mask (*pANDplane*), each bit set to zero is cleared. Each bit set to one remains unchanged. Each one bit in the XOR mask (*pXORplane*) is then inverted. Rather than using masks, cursors can be loaded from a resource script using the *LoadCursor()* function.

DestroyCursor() destroys a cursor. The hCursor parameter contains the handle of the cursor to be destroyed.

94.3 Returns

CreateCursor() returns the handle of the new cursor if successful. Otherwise, it returns NULL.

DestroyCursor() returns TRUE, if it is successful. Otherwise, it returns FALSE.

94.4 Errors

None.

94.5 Cross-References

LoadCursor()

---

95 LoadCursor

95.1 Synopsis

HCURSOR LoadCursor(HINSTANCE hInstance, LPCSTR pszName);

95.2 Description

The *LoadCursor()* function loads a cursor from an application or DLL resource script. The *hInstance* parameter holds the instance handle of the executable containing the cursor. This can be an application's instance handle, the instance handle of a Dynamic Link Library (DLL), or NULL to load one of the system cursors. The *pszName* parameter contains a pointer to a character string containing the name of the cursor resource.

If the cursor resource has a numerical ID, this string can be of the form "#*nn*", where *nn* is the resource ID. Or the application can use the MAKEINTRESOURCE macro to form a pointer from the resource ID.

DestroyCursor() must be used to destroy this cursor.

95.3 Returns

This function returns the handle of the cursor, if it is successful.

If *LoadCursor()* cannot find the resource, it returns NULL. If the specified resource ID actually references a different type of resource, such as a dialog box, *LoadCursor()* returns a value, but this value will be invalid.

95.4 Errors

None.

95.5 Cross-References

None.

---

96 GetCursorPos, SetCursorPos

96.1 Synopsis

void GetCursorPos(POINT *pPoint);

void SetCursorPos(int x, int y);

96.2 Description

The *GetCursorPos()* function determines the current position of the cursor. The *pPoint* parameter is a pointer to a **POINT** structure that receives the coordinates of the cursor.

SetCursorPos() changes the cursor's current position. The cursor's new position is determined by the *x* and *y* parameters.

96.3 Returns

GetCursorPos() returns the current cursor position in the structure pointed to by the *pPoint* parameter. *SetCursorPos()* does not return any value.

96.4 Errors

None.

96.5 Cross-References

None.

---

97 ShowCursor

97.1 Synopsis

int ShowCursor(BOOL fShow);

97.2 Description

The *ShowCursor()* function shows and hides the cursor. The *fShow* flag tells whether to show or hide the cursor. If the *fShow* parameter is TRUE, the cursor is shown. If it is FALSE, the cursor is hidden.

Windows maintains an internal display count for the cursor. *ShowCursor()* increments the display count if the *fShow* parameter is TRUE, and decrements it if *fShow* is FALSE.

97.3 Returns

ShowCursor() returns the new display count of the window. The cursor is visible if the display count is greater than or equal to zero.

97.4 Errors

None.

97.5 Cross-References

None.

---

98 GetCursor, SetCursor

98.1 Synopsis

HCURSOR GetCursor(void);

HCURSOR SetCursor(HCURSOR hCursor);

98.2 Description

The *GetCursor()* function returns the handle of the current cursor. It uses no parameters.

The *SetCursor()* function changes the current cursor image. The *hCursor* parameter is the handle of the new cursor. The cursor can be created using the *CreateCursor()* function or loaded from a resource script using the *LoadCursor()* function.

98.3 Returns

GetCursor() returns the handle of the current cursor.

SetCursor() returns the handle of the last cursor, or NULL if there was no previous cursor.

98.4 Errors

None.

98.5 Cross-References

CreateCursor(), LoadCursor()

99 ClipCursor

99.1 Synopsis

void ClipCursor(const RECT *pRect);

99.2 Description

The *ClipCursor()* function constrains the cursor to the specified region on the screen. The *pRect* parameter contains a pointer to a **RECT** structure containing the bounding rectangle of this region.

99.3 Returns

None.

99.4 Errors

None.

99.5 Cross-References

None.

100 GetClipCursor

100.1 Synopsis

void GetClipCursor(RECT *lprc);

100.2 Description

The *GetClipCursor()* function returns the screen coordinates of the cursor's bounding rectangle after the previous call to the *ClipCursor()* function. If the cursor is not confined to a rectangle, the function returns the dimensions of the screen.

100.3 Returns

None.

100.4 Errors

None.

100.5 Cross-References

ClipCursor(), SetCursorPos()

101 CopyCursor

101.1 Synopsis

int CopyCursor(HINSTANCE hinst, HCURSOR hcur);

101.2 Description

The *CopyCursor()* function makes a copy of the cursor. The *hcur* parameter specifies the cursor to be copied.

101.3 Returns

If successful, the *CopyCursor()* function returns the handle of the duplicate cursor. Otherwise, it returns NULL.

101.4 Errors

None.

101.5 Cross-References

DestroyCursor(), GetCursor(), SetCursor(), ShowCursor()

102 SetProp, GetProp

102.1 Synopsis

BOOL SetProp(HWND hWnd, LPCSTR lpsz, HANDLE hData);

HANDLE GetProp(HWND hWnd, LPCSTR lpsz);

102.2 Description

The *SetProp()* and *GetProp()* functions add and retrieve entries from the property list of the window specified by the *hWnd* parameter. *SetProp()* adds or modifies the property specified by *lpsz*. The data handle *hData* identifies the data that is copied to the property list. *GetProp()* retrieves the data handle from the property list of the window *hWnd*.

The property specified by *lpsz* may be a zero-terminated string or a global atom. If *lpsz* is a global atom it must be a 16-bit value placed in the low-order word.

102.3 Returns

SetProp() returns TRUE if the string and data handle are added to the property list. Otherwise, it returns FALSE.

GetProp() returns the data handle stored in the property list. Otherwise, it returns zero.

102.4 Errors

None.

102.5 Cross-References

RemoveProp()

103 RemoveProp

103.1 Synopsis

HANDLE RemoveProp(HWND hWnd, LPCSTR lpsz);

103.2 Description

The *RemoveProp()* function removes an entry from the property list of the window specified by the *hWnd* parameter. The property specified by lpsz may be a zero-terminated string or a global atom. If *lpsz* is a global atom it must be a 16-bit value placed in the low-order word.

103.3 Returns

RemoveProp() returns the data handle stored in the property list, if it is successful. It returns zero, if it is unsuccessful.

103.4 Errors

None.

103.5 Cross-References

SetProp(), GetProp()

104 EnumProps

104.1 Synopsis

int EnumProps(HWND hWnd, PROPENUMPROC propenmproc);

104.2 Description

The *EnumProps()* function enumerates all entries in the property list of the window specified by the *hWnd* parameter. Each entry is passed one by one to the specified call-back function *propenmproc*. The call-back function *propenmproc* is application defined. An example is given in *EnumPropsProc()*.

104.3 Returns

The *EnumProps()* function returns the last value returned by the callback function, if it is successful. Otherwise, it returns -1.

104.4 Errors

None.

104.5 Cross-References

SetProp(), GetProp(), RemoveProp(), MyEnumPropsProc()

105 EnumPropsProc

105.1 Synopsis

BOOL MyEnumPropsProc(HWND hWnd, LPCSTR lpsz, HANDLE hData);

105.2 Description

The *EnumPropsProc()* function is an example of the application-defined callback function used by *EnumProps()*. The callback is called one time for each property in the property list of the window *hWnd*.

The property identified by lpsz may be a zero-terminated string or a global atom. If *lpsz* is a global atom,

The *hData* parameter will be the associated data for the identified property.

105.3 Returns

EnumPropsProc() must return TRUE to continue enumerating the property list. If it returns FALSE, the enumeration discontinues.

105.4 Errors

None.

105.5 Cross-References

EnumProps()

106 GetClipboardViewer

106.1 Synopsis

HWND GetClipboardViewer(void);

106.2 Description

When the contents of the clipboard is modified, a WM_DRAWCLIPBOARD message is sent to each window contained in a list (chain) maintained for the clipboard.

GetClipboardViewer() can be called by the application to get the handle of the first window in the clipboard's window chain.

106.3 Returns

If the *GetClipboardView()* function is successful, it returns the handle of the first window in the clipboard window chain.

If there is no window in the clipboard window chain, the *GetClipboardViewer()* function returns NULL.

106.4 Errors

None.

106.5 Cross-References

WM_CHANGECBCHAIN, WM_DRAWCLIPBOARD

107 SetClipboardViewer

107.1 Synopsis

HWND SetClipboardViewer(HWND hWnd);

107.2 Description

When the contents of the clipboard is modified, the WM_DRAWCLIPBOARD is sent to each window contained in a list (chain) maintained for the clipboard.

SetClipboardViewer() is called by the application to add a window to the clipboard's window chain. The *hWnd* parameter contains the handle of the window that is added to the clipboard's window chain.

107.3 Returns

If the *SetClipboardViewer()* function is successful, it returns the handle of the window that follows the inserted window in the clipboard's window chain.

107.4 Errors

None.

107.5 Cross-References

WM_CHANGECBCHAIN

108 ChangeClipboardChain

108.1 Synopsis

BOOL ChangeClipboardChain(HWND hWnd, HWND hNextWnd);

108.2 Description

When the contents of the clipboard is modified, the WM_DRAWCLIPBOARD is sent to each window contained in a list (chain) maintained for the clipboard.

ChangeClipboardChain() is called by an application to remove a window from the clipboard's window chain. The *hWnd* parameter contains the handle of the window that is removed from the clipboard's window chain. The *hNextWnd* parameter contains the handle of the window that is returned by *the SetClipboardViewer()* function when *hWnd* is added to the window chain.

108.3 Returns

If the *ChangeClipboardChain()* function is successful, it returns TRUE. If the *ChangeClipboardChain()* function is not successful, it returns FALSE.

108.4 Errors

None.

108.5 Cross-References

WM_CHANGECBCHAIN

109 OpenClipboard

109.1 Synopsis

BOOL OpenClipboard(HWND hWnd);

109.2 Description

The *OpenClipboard()* function opens the clipboard and prevents other applications from modifying it until it is closed. The *hWnd* parameter contains the window handle that is associated with the open clipboard. The window does not become the owner of the clipboard until the *EmptyClipboard()* function is called.

109.3 Returns

If the *OpenClipboard()* function is successful, it returns TRUE. If another application or window already has the clipboard open, the function returns FALSE.

109.4 Errors

None.

109.5 Cross-References

CloseClipboard(), EmptyClipboard(), GetClipboardOwner(), GetOpenClipboardWindow()

---

110 GetOpenClipboardWindow

110.1 Synopsis

HWND GetOpenClipboardWindow(void);

110.2 Description

The *GetOpenClipboardWindow()* function returns the handle of the window that is associated with an open clipboard.

110.3 Returns

If the *GetOpenClipboardWindow()* function is successful, it returns the handle of the window that is associated with an open clipboard. If the clipboard is closed, the *GetOpenClipboardWindow()* function returns NULL.

110.4 Errors

None.

110.5 Cross-References

CloseClipboard(), EmptyClipboard(), GetClipboardOwner(), OpenClipboard()

---

111 CloseClipboard

111.1 Synopsis

BOOL CloseClipboard(void);

111.2 Description

The *CloseClipboard()* function closes the clipboard and allows other applications to open the clipboard for their own use.

111.3 Returns

If the *CloseClipboard()* function is successful, it returns TRUE. If the *CloseClipboard()* function is not successful, it returns FALSE.

111.4 Errors

None.

111.5 Cross-References

EmptyClipboard(), GetClipboardOwner(), GetOpenClipboardWindow(), OpenClipboard()

---

112 EmptyClipboard

112.1 Synopsis

BOOL EmptyClipboard(void);

112.2 Description

The *EmptyClipboard()* function frees any handle associated with the data in an open clipboard and assigns the ownership of the clipboard to the window that is associated with the open clipboard. The clipboard must be successfully opened by the *OpenClipboard()* function before the *EmptyClipboard()* function is called.

112.3 Returns

If the *EmptyClipboard()* function is successful, it returns TRUE. If the *EmptyClipboard()* function is not successful, it returns FALSE.

112.4 Errors

None.

112.5 Cross-References

CloseClipboard(), EmptyClipboard(), GetClipboardOwner(), OpenClipboard()

---

113 GetClipboardOwner

113.1 Synopsis

HWND GetClipboardOwner(void);

113.2 Description

The *GetClipboardOwner()* function returns the handle of the window that owns the clipboard. The clipboard is assigned an owner when the *EmptyClipboard()* function is called.

113.3 Returns

If the *GetClipboardOwner()* function is successful, it returns the handle of the window that owns the clipboard. If the clipboard is not owned by a window, the *GetClipboardOwner()* function returns NULL.

113.4 Errors

None.

113.5 Cross-References

CloseClipboard(), EmptyClipboard(), GetOpenClipboardWindow(), OpenClipboard()

---

114 CountClipboardFormats

114.1 Synopsis

int CountClipboardFormats(void);

114.2 Description

The *CountClipboardFormats()* function returns the number of clipboard formats used by the data in the clipboard.

114.3 Returns

CountClipboardFormats() returns the number of clipboard formats used by the data in the clipboard.

114.4 Errors

None.

114.5 Cross-References

EnumClipboardFormats(), GetPriorityClipboardFormat(), IsClipboardFormatAvailable()

---

115 EnumClipboardFormats

115.1 Synopsis

UINT EnumClipboardFormats(UINT FormatID);

115.2 Description

The *EnumClipboardFormats()* function returns the format identifiers for data in the clipboard. If the *FormatID* parameter's value is NULL, the function returns the first format identifier in the list of format identifiers. If the *FormatID* parameter's value is not NULL, the function assumes that the *FormatID* parameter is an identifier found in the list, looks for that value in the list and returns the value of the next format identifier in the list. Therefore, to get all of the format identifiers in the list, successive calls can be made to the *EnumClipboardFormats()* function using the function's last return value. The clipboard must be opened by the *OpenClipboard()* function before calling the *EnumClipboardFormats()* function.

115.3 Returns

If the *EnumClipboardFormats()* function is successful, it returns the next format identifier for data in the clipboard. If the *EnumClipboardFormats()* function is not successful, it returns zero.

115.4 Errors

None.

115.5 Cross-References

CountClipboardFormats(), GetPriorityClipboardFormat(), IsClipboardFormatAvailable()

---

116 GetPriorityClipboardFormat

116.1 Synopsis

int GetPriorityClipboardFormat(UINT *FormatList, int Count);

116.2 Description

The *GetPriorityClipboardFormat()* function accepts a list of clipboard format identifiers and returns the identifier of the first clipboard format for which data exists in the clipboard. The *FormatList* parameter contains a pointer to the list of format identifiers and the *Count* parameter is the number of identifiers in *FormatList*.

116.3 Returns

GetPriorityClipboardFormat() returns the identifier of the first clipboard format for which data exists in the clipboard. If the clipboard is empty, the function returns NULL. If the function cannot find any data in the clipboard that matches the clipboard formats in the given list, it returns -1.

116.4 Errors

None.

116.5 Cross-References

CountClipboardFormats(), EnumClipboardFormats(), IsClipboardFormatAvailable()

---

117 IsClipboardFormatAvailable

117.1 Synopsis

BOOL IsClipboardFormatAvailable(UINT FormatID);

117.2 Description

The *IsClipboardFormatAvailable()* function determines whether the clipboard contains data of a specific clipboard data format. The desired format's identifier is supplied to the function in the *FormatID* parameter.

117.3 Returns

IsClipboardFormatAvailable() returns TRUE if the clipboard contains data of the specified format. The function returns FALSE if the clipboard does not contain data of the specified format.

117.4 Errors

None.

117.5 Cross-References

CountClipboardFormats(), EnumClipboardFormats(), GetPriorityClipboardFormat()

118 RegisterClipboardFormat, GetClipboardFormatName

118.1 Synopsis

UINT RegisterClipboardFormat(LPCSTR FormatName);

int GetClipboardFormatName(UINT FormatID, LPSTR FormatName, int Count);

118.2 Description

The *RegisterClipboardFormat()* function registers a new clipboard format using the format name supplied in the function's *FormatName* parameter. The *GetClipboardFormatName()* function copies the name of a registered format into a buffer. The *FormatID* parameter should contain the identifier of the clipboard format. The name of the format is copied into the buffer given in the *FormatName* parameter. The *Count* parameter contains the length of the buffer in bytes.

118.3 Returns

If the *RegisterClipboardFormat()* function is successful, it returns a non-zero value in the range of 0xC000 through 0xFFFF. The function's return value is an identifier associated with the registered format. If the format name has already been registered by the calling application or a different application, the function returns the same identifier value that it returned when it originally registered the format name. If the *RegisterClipboardFormat()* function is not successful, the function returns zero.

GetClipboardFormatName() returns the number of bytes of the format name copied into the given buffer. If the format name is predefined or has not been registered, *GetClipboardFormatName()* returns zero.

118.4 Errors

None.

118.5 Cross-References

None.

119 GetClipboardData, SetClipboardData

119.1 Synopsis

HANDLE SetClipboardData(UINT Format, HANDLE hData);

HANDLE GetClipboardData(UINT Format);

119.2 Description

The *SetClipboardData()* function places clipboard data into an opened clipboard. A handle to the clipboard data is passed in the *hData* parameter and the clipboard format of the data is specified in the *Format* parameter. If *hData* is a handle to memory allocated using the *GlobalAlloc()* function, the handle must not be used after it is passed to the *SetClipboardData()* function. If the value of the *hData* parameter is NULL, the owner of the clipboard is sent a WM_RENDERFORMAT message when the clipboard data must be supplied. The data placed in the clipboard by the *SetClipboardData()* function is not shown in the clipboard's window until the *CloseClipboard()* function is called.

The *GetClipboardData()* function returns the handle for clipboard data of a specfic type that is in the clipboard. The clipboard must be open before calling this function. The clipboard format of the data is specified in the *Format* parameter. The data handle returned by the function is owned by the clipboard. After getting the data handle, an application should promptly copy the data and release the handle. The data handle should not be freed or left in a locked state. If a request for text in the CF_TEXT text format is made and the clipboard only contains text in the CF_OEMTEXT text format, the data is converted to CF_TEXT text format and vice-versa. When the clipboard contains data with the CF_PALETTE format, assume that the clipboard's other data has been realized against that palette.

The following predefined clipboard formats are recognized:

CF_BITMAP	This value specifies a handle of type HBITMAP.
CF_DIB	This value specifies a handle allocated by the *GlobalAlloc()* function; the data contains a **BITMAPINFO** structure followed by the bitmaps data.
CF_DIF	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in the Data Interchange Format (DIF).
CF_DSPBITMAP	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in a private format and represents a bitmap. The data is displayed using the bitmap format.
CF_DSPMETAFILEPICT	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in a private format and represents a metafile. The data is displayed using the metafile format.
CF_DSPTEXT	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in a private format and represents text. The data is displayed using the text format.
CF_METAFILEPICT	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is a metafile.
CF_OEMTEXT	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is an array of text characters using the OEM character set. A carriage return and linefeed in the array signifies a new line. A null character in the array signifies the end of the data.
CF_OWNERDISPLAY	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in a private format and is of an unknown type that is displayed by the owner of the clipboard.
CF_PALETTE	This value specifies a handle of type HPALETTE.
CF_PENDATA	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is for the pen extensions to the Windows operating system.
CF_RIFF	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in the (RIFF) Resource Interchange File Format.
CF_SYLK	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in the (SYLK) Microsoft Symbolic Link format.
CF_TEXT	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is an array of text characters; a carriage return and linefeed in the array signifies a new line; a null character in the array signifies the end of the data.
CF_TIFF	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is in the (TIFF) Tag Image File Format.
CF_WAVE	This value specifies a handle allocated by the *GlobalAlloc()* function; the data is sound wave information from a RIFF WAVE sound file.
CF_PRIVATEFIRST - CF_PRIVATELAST	This value specifies a handle allocated by the *GlobalAlloc()* function; when the data is removed from the clipboard, the WM_DESTROYCLIPBOARD message is used to free the data.
CF_GDIOBJFIRST - CF_GDIOBJLAST	This value specifies a handle allocated by the *GlobalAlloc()* function.

When the data is removed from the keyboard, it can be freed by calling the *DeleteObject()* function.

119.3 Returns

If the *SetClipboardData()* function is successful, it returns the handle of the data. If the *SetClipboardData()* function is not successful, it returns NULL.

If the *GetClipboardData()* function is successful, it returns a handle to clipboard data of the specified format. If the *GetClipboardData()* function is not successful, it returns NULL.

119.4 Errors

None.

119.5 Cross-References

None.

120 CallWndProc

120.1 Synopsis

LRESULT CALLBACK CallWndProc(int codeflag, WPARAM wParam, LPARAM lParam);

120.2 Description

The *CallWndProc()* function is called by the system whenever a *SendMessage()* function is called. It passes the message to the callback function before passing the message to the destination window procedure. Note that this function is a library-defined callback function. The *codeflag* parameter identifies whether the callback function is to process the message or call the *CallNextHookEx()* function.

If the value of this parameter is less than zero, the message is passed on to the *CallNextHookEx()* function without further processing. If the *wParam* parameter has a non-zero value the message is sent by the current task. Otherwise, the value is NULL. The *lParam* parameter contains a pointer to the structure that has the message data. The list below shows the each of the different members of the structure and its description.

lParam	This member contains the *lParam* parameter of the message .
wParam	This member contains the *wParam* parameter of the message.
uMsg	This member specifies the message.
hWnd	This member identifies the window that will receive the message.

CallWndProc() can examine or modify the contents of the message before returning control to the system. The message is then passed to the window procedure. Note that this function should be in a dynamic-link library. Since *CallWndProc()* is a placeholder name for the library-defined function name, the EXPORTS statement in the library's module-definition file should have the actual name that is exported. An application that wishes to call this function must first install the callback function by specifying the WH_CALLWNDPROC filter type and the procedure-instance address of the callback function in a call to the *SetWindowsHookEx()* function.

120.3 Returns

A zero should be returned by the callback function CallWndProc().

120.4 Errors

None.

120.5 Cross-References

CallNextHookEx(), SendMessage(), SetWindowsHookEx()

121 GetMsgProc

121.1 Synopsis

LRESULT CALLBACK GetMsgProc(int codeflag, WPARAM wParam, LPARAM lParam);

121.2 Description

The GetMsgProc() function is called by the system when a GetMessage() function has fetched a message from the application queue. This message is passed to the callback function before the message is passed to the destination window procedure. This function is a library-defined callback function. The codeflag parameter identifies whether the callback function should process the message or call the function *CallNextHookEx()*. If the value of this parameter is less than zero, the callback function should pass the message to the *CallNextHookEx()* without doing any processing. The *wParam* parameter is set to NULL. The *lParam* parameter contains a pointer to the **MSG** structure, which holds the message information.

GetMsgProc() examines or modifies the contents of the message before returning control to the system. At this time the message, with modifications if any, is passed to the window procedure. Note that this function should be in a dynamic-link library. The callback functions do not require a return value. Since the *GetMsgProc()* is a placeholder name for the library-defined function name, the EXPORTS statement in the library's module-definition file should have the actual name to be exported. An application that calls this function must first install the callback function by specifying the WH_GETMESSAGE filter type and the procedure-instance address of the callback function in a call to the *SetWindowsHookEx()* function.

121.3 Returns

The function *GetMsgProc()* returns zero.

121.4 Errors

None.

121.5 Cross-References

CallNextHookEx(), GetMessage(), SetWindowsHookEx()

---

122 MessageProc

122.1 Synopsis

LRESULT CALLBACK MessageProc(int codeflag, WPARAM wParam, LPARAM lParam);

122.2 Description

The *MessageProc()* function is called by the system after a dialog box, message box, or menu has fetched a message that has not been processed. This callback function can modify or process these messages. The *codeflag* parameter identifies the type of message that is being processed. This parameter can have one of the following values:

MSGF_DIALOGBOX	Messages inside a dialog box or message box procedure are being processed.
MSGF_MENU	Keyboard and mouse messages in a menu are being processed.

If the value of this parameter is less than zero, this function passes the message transparently to the function *CallNextHookEx()* and returns a value of the *CallNextHookEx()* function. The *wParam* parameter should be set to NULL.

The *lParam* parameter contains a pointer to the **MSG** message structure that contains the message information.

Since *MessageProc()* is a placeholder name for the library-defined function name, the EXPORTS statement in the library's module-definition file should have the actual name to be exported. An application that calls this function must first install the callback function by specifying the WH_SYSMSGFILTER filter type and the procedure-instance address of the callback function in a call to the *SetWindowsHookEx()* function.

122.3 Returns

The function *MessageProc()* returns a non-zero value if the message is processed by the function. Otherwise, it returns zero.

122.4 Errors

None.

122.5 Cross-References

CallNextHookEx(), SetWindowsHookEx()

123 SysMsgProc

123.1 Synopsis

LRESULT CALLBACK SysMsgProc(int codeflag, WPARAM wParam, LPARAM lParam);

123.2 Description

The *SysMsgProc()* function is called by the system after a dialog box, message box, or menu has fetched a message but before the message has been processed. This callback function can modify or process messages for any application in the system. The *codeflag* parameter identifies the type of message that is being processed. This parameter can have one of the following values:

MSGF_DIALOGBOX	Messages inside a dialog box or message box procedure are being processed.
MSGF_MENU	Keyboard and mouse messages in a menu are being processed.

If the value of this parameter is less than zero, this function passes the message transparently to the function *CallNextHookEx()* and returns back the return value of the *CallNextHookEx()* function. The *wParam* parameter should be set to NULL.

The *lParam* parameter contains a pointer to the **MSG** message structure that contains the message information.

Note that this function should be in a dynamic-link library. Since the *SysMsgProc()* is a placeholder name for the library-defined function name, the EXPORTS statement in the library's module-definition file should have the actual name to be exported. An application that calls this function must first install the callback function by specifying the WH_SYSMSGFILTER filter type and the procedure-instance address of the callback function in a call to the *SetWindowsHookEx()* function.

123.3 Returns

The function *SysMsgProc()* returns a non-zero value if the message is processed by the function. Otherwise, it returns zero.

123.4 Errors

None.

123.5 Cross-References

CallNextHookEx(), MessageBox(), SetWindowsHookEx()

124 AddAtom, GlobalAddAtom

124.1 Synopsis

ATOM WINAPI AddAtom(LPCSTR lpcstr);

ATOM WINAPI GlobalAddAtom(LPCSTR lpcstr);

124.2 Description

The *AddAtom()* and *GlobalAddAtom()* functions add the null-terminated string pointed to by *lpcstr* to the local and global atom tables respectively. The functions return a reference value that uniquely identifies the string.

If a particular string is already in the atom table, its reference count is incremented and the reference value is returned. This ensures that only one copy of a particular string is stored, reducing memory requirements.

Atoms are not case sensitive.

124.3 Returns

These functions return a value that uniquely identifies the atom, if the functions are successful. These functions return zero, if not successful.

124.4 Errors

None.

124.5 Cross-References

DeleteAtom(), FindAtom(), GetAtomName()

---

125 DeleteAtom, GlobalDeleteAtom

125.1 Synopsis

ATOM WINAPI DeleteAtom(ATOM atm);

ATOM WINAPI GlobalDeleteAtom(ATOM atm);

125.2 Description

The *DeleteAtom()* and *GlobalDeleteAtom()* functions decrement the reference count of the atom specified by the value, *atm*. When an atom's reference count is zero, the string is removed from the corresponding atom table.

125.3 Returns

These functions return zero, if the function is successful. If the functions are unsuccessful, they return the atom value atm.

125.4 Errors

None.

125.5 Cross-References

AddAtom()

---

126 FindAtom, GlobalFindAtom

126.1 Synopsis

ATOM WINAPI FindAtom(LPCSTR lpcstr);

ATOM WINAPI GlobalFindAtom(LPCSTR lpcstr);

126.2 Description

The *FindAtom()* and *GlobalFindAtom()* functions search their respective atom tables for the string specified by *lpcstr*. If the string is found, its reference value is returned.

126.3 Returns

These functions return the reference value of the atom matching the *lpcstr*. If a match is not found, zero is returned.

126.4 Errors

None.

126.5 Cross-References

AddAtom(), DeleteAtom(), GetAtomName()

---

127 GetAtomName, GetGlobalAtomName

127.1 Synopsis

UINT WINAPI GetAtomName(ATOM atm, LPSTR lpszBuffer, int cbBuffSize);

UINT WINAPI GlobalGetAtomName(ATOM atm, LPSTR lpszBuffer, int cbBuffSize);

127.2 Description

The *GetAtomName()* and *GlobalGetAtomName()* functions get a copy of the atom's string, referenced by atm, from their respective atom tables. The string is copied to the buffer pointed to by *lpszBuffer*. The total number of bytes copied is limited by the *cbBuffSize* parameter.

127.3 Returns

The return value is the actual number of bytes copied to the buffer pointed to by *lpszBuffer*.

127.4 Errors

None.

127.5 Cross-References

AddAtom(), DeleteAtom(), FindAtom ()

---

128 InitAtomTable

128.1 Synopsis

BOOL WINAPI InitAtomTable(int cTableEntries);

128.2 Description

The *InitAtomTable()* function initializes the local atom table to the number of entries specified by *cTableEntries*. The default size of the atom table is 37 entries. This function does not need to be called to use the local atom table, however, it does provide the application with the ability to allocate a larger atom table, if needed. By increasing the size of the table, additions to the atom table will be faster.

128.3 Returns

The return value is TRUE if *InitAtomTable()* is successful. Otherwise, it returns FALSE.

128.4 Errors

None.

128.5 Cross-References

AddAtom()

---

129 CheckDlgButton

129.1 Synopsis

void CheckDlgButton(HWND hDlg, int nButton, UINT Check);

129.2 Description

The *CheckDlgButton()* function is used to set the state of a button control. The button control is specified by its ID and dialog box in the *nButton* and *hDlg* parameters, respectively. The *Check* parameter indicates the new state of the button:

0	The button is not selected.
1	The button is selected.
2	The button is disabled (if it is a three-state button).

The *CheckDlgButton()* function sends a BM_SETCHECK message to the specified button control.

129.3 Returns

None.

129.4 Errors

None.

129.5 Cross-References

CheckRadioButton(), *IsDlgButtonChecked()*, BM_GETCHECK, BM_SETCHECK

---

130 CheckRadioButton

130.1 Synopsis

void CheckRadioButton(HWND hDlg, int nFirst, int nLast, int nCheck);

130.2 Description

The *CheckRadioButton()* function checks one radio button from a group and removes check marks from any other radio buttons in the group. The *hDlg* parameter contains the handle of the dialog box containing the radio button group. The *nCheck* parameter contains the ID of the radio button which receives the check mark. The *nFirst* and *nLast* parameters contain the ID of the first and last radio buttons in the group. This function sends a BM_SETCHECK message to the radio button control.

130.3 Returns

None.

130.4 Errors

None.

130.5 Cross-References

CheckDlgButton(), IsDlgButtonChecked(),BM_SETCHECK

---

131 CreateDialog, CreateDialogIndirect

131.1 Synopsis

```
HWND CreateDialog(HINSTANCE hInstance, LPCSTR lpTemplate, HWND hOwner,
        DLGPROC DlgProc);
HWND CreateDialogIndirect(HINSTANCE hInstance, void *lpTemplate, HWND hOwner,
        DLGPROC DlgProc);
```

131.2 Description

The *CreateDialog()* and *CreateDialogIndirect()* functions are used to launch a modeless dialog box. These functions return after the dialog box has been created. The dialog box can thereafter be destroyed by using the *DestroyWindow()* function.

The instance handle of the module creating the dialog box is provided in the *hInstance* parameter. For *CreateDialog()*, the dialog box template is specified by its resource name in the *lpTemplate* parameter. For *CreateDialogIndirect()*, the template address is specified in the *lpTemplate* parameter. The owner window of the dialog box is specified in *hOwner*. The address of the message handling routine for the dialog box is specified by the *DlgProc* parameter. This address is created by using the *MakeProcInstance()* function.

The dialog box is created by calling the *CreateWindowEx()* function. If the DS_SETFONT style is specified, a WM_SETFONT message is sent before the WM_INITDIALOG message to the message handling routine. If the WS_VISIBLE style is specified, the dialog box appears in the owner window.

131.3 Returns

If successful, these functions return a handle to the created dialog box. Otherwise, they return NULL.

131.4 Errors

None.

131.5 Cross-References

CreateDialogIndirectParam(), CreateDialogParam(), DestroyWindow(), MakeProcInstance(), WM_INITDIALOG, WM_SETFONT, DS_SETFONT, WS_VISIBLE

---

132 CreateDialogParam, CreateDialogIndirectParam

132.1 Synopsis

```
HWND CreateDialogParam(HINSTANCE hInstance, LPCSTR lpszTemplate, HWND hwndOwner,
        DLGPROC dlgProc, LPARAM lParam);
HWND CreateDialogParamIndirect(HINSTANCE hInstance, const void *pTemplate,
```

HWND hwndOwner, DLGPROC dlgProc, LPARAM lParam);

132.2 Description

The *CreateDialogParam()* and *CreateDialogParamIndirect()* functions create a modeless dialog, allowing a value to be passed to the dialog procedure in the WM_INITDIALOG message. *CreateDialogParam()* loads a dialog template from an application or DLL resource table. *CreateDialogParamIndirect()* creates the dialog from a template already in memory.

The *hInstance* parameter of both functions contains the instance handle of the module creating the dialog. If *CreateDialogParam()* is to create a dialog from a module besides the currently executing one, such as a template in a DLL, the *hInstance* parameter must contain the instance handle of the module whose resource table contains the dialog template. The *lpszTemplate* parameter of *CreateDialogParam()* contains the name of the dialog template in the resource table. The *pTemplate* parameter of *CreateDialogParamIndirect()* contains a pointer to the dialog template in memory. This parameter must point to a **DialogBoxHeader** structure. The *hwndOwner* parameter of both functions contains the window handle of the parent window of the modeless dialog. The *dlgProc* parameter contains a pointer to the dialog's dialog procedure. See the *DialogProc()* callback for more information. The *lParam* parameter contains an arbitrary 32-bit value. This value will be passed to the dialog procedure as the *lParam* value of the WM_INITDIALOG message.

132.3 Returns

If successful, these functions return the window handle of the newly created dialog box. If they fail, they return NULL.

132.4 Errors

None.

132.5 Cross-References

WM_INITDIALOG, *DialogProc()*

133 DialogBox, DialogBoxIndirect

133.1 Synopsis

int DialogBox(HINSTANCE hInstance, LPSTR Template, HWND hOwner, DLGPROC DlgProc);

int DialogBoxIndirect(HINSTANCE hInstance, HANDLE Template, HWND hOwner,

DLGPROC DlgProc);

133.2 Description

The *DialogBox()* and *DialogBoxIndirect()* functions create modal dialog boxes. The *hInstance* parameter identifies the module containing the dialog box template. The *hOwner* parameter is the handle of the owner window. The *DlgProc* parameter is the address of the dialog box callback procedure. For the *DialogBox()* function, *Template* is a string pointer to the name of the dialog box template. The *DialogBoxIndirect()* function interprets the *Template* parameter as a handle to a memory address containing the dialog box template in the form of a *DialogBoxHeader* structure. These functions call *CreateWindowEx()* to create the dialog box and do not return control until the *EndDialog()* function is called from the dialog box callback procedure.

133.3 Returns

The return value is generated from within dialog box callback procedure. When the *EndDialog()* function is called, the *Result* parameter becomes the return value for function which created the dialog box. If the dialog box cannot be created, the return value is -1.

133.4 Errors

None.

133.5 Cross-References

DialogBoxIndirectParam(), DialogBoxParam(), DialogProc(), EndDialog(), CreateWindowEx()

134 DialogBoxIndirectParam, DialogBoxParam

134.1 Synopsis

int DialogBoxIndirectParam(HINSTANCE hInstance, HANDLE Template, HWND hOwner,

DLGPROC DlgProc, LPARAM lParam);

int DialogBoxParam(HINSTANCE hInstance, LPSTR Template, HWND hOwner, DLGPROC DlgProc,

LPARAM lParam);

134.2 Description

The *DialogBoxParam()* and *DialogBoxIndirectParam()* functions create modal dialog boxes with an initialization parameter. The *hInstance* parameter identifies the module containing the dialog box template. The *hOwner* parameter is the handle of the owner window. The *DlgProc* parameter is the address of the dialog box callback procedure. The *lParam* parameter is a long value that is forwarded to the *DlgProc* procedure when called with the WM_INITDIALOG message. For the *DialogBoxParam()* function, *Template* is a string pointer to the name of the dialog box template. *DialogBoxIndirectParam()* interprets the *Template* parameter as a handle to a memory address containing the dialog box template in the form of a **DLGTEMPLATE** structure. These functions call *CreateWindowEx()* to create the dialog box and do not return control until the *EndDialog()* function is called from the dialog box callback procedure.

134.3 Returns

The return value is generated from within the dialog box callback procedure. When the *EndDialog()* function is called, the *Result* parameter becomes the return value for function which created the dialog box. If the dialog box cannot be created, the return value is -1.

134.4 Errors

None.

134.5 Cross-References

DialogBox(), DialogBoxIndirect(), DialogProc(), EndDialog(), CreateWindowEx(), WM_INITDIALOG

135 DlgDirList, DlgDirListComboBox

135.1 Synopsis

int DlgDirList(HWND hwndDlg, LPSTR lpszPath, int idListBox, int idStatic, UINT uAttrib);

int DlgDirListComboBox(HWND hwndDlg, LPSTR lpszPath, int idComboBox, int idStatic,

UINT uAttrib);

135.2 Description

The *DlgDirList()* function fills controls in a dialog box with information about a directory. The *hwndDlg* parameter contains the handle of the dialog box or window that contains the controls that will receive the information. The *lpszPath* parameter is a pointer to a string containing the directory to change to, before displaying information. If this string contains wildcards specifying certain files to list, such as "*.exe", only those files will be placed in the list box. The string will be updated after the list box is filled by removing either the drive or directory, or both, from the given pathname. The *idListBox* parameter contains the dialog-control identifier of the list box to be filled with file names from the specified directory. The *idStatic* parameter contains the dialog-control identifier of a static control whose text will be changed to the specified path.

The *uAttrib* parameter contains a bitwise OR of several flags indicating which files will be listed. If this bitmask contains DDL_READWRITE, all read-write files with no other attributes will be listed. If it contains DDL_READONLY, all read-only files will be listed. DDL_HIDDEN causes hidden files to be listed; DDL_SYSTEM lists system files; DDL_DIRECTORY lists directories; DDL_ARCHIVE lists archived files; DDL_DRIVES lists drives. If the bitmask contains DDL_EXCLUSIVE, only the specified files are listed. If this flag is absent, all normal files are listed in addition to the specified files. If it contains DDL_POSTMSGS, the *PostMessage()* API will be used to send messages through the application's message queue. Normally, the *SendMessage()* API is used to send messages directly to the dialog controls.

If directories are added to the list box, their names are in brackets, such as "[windows]". If drive names are added to the list box, their names are enclosed in brackets and hyphens, such as "[-c-]".

The *DlgDirListComboBox()* function fills controls in a dialog box with information on a directory. It is identical to the *DlgDirList()* function, except that it places the list of files in a combo box rather than a list box.

135.3 Returns

DlgDirList() and *DlgDirListComboBox()* return a non-zero value, if they are successful. Otherwise, they return zero.

135.4 Errors

None.

135.5 Cross-References

PostMessage(), SendMessage()

136 DlgDirSelect, DlgDirSelectComboBox

136.1 Synopsis

BOOL DlgDirSelect(HWND hwndDlg, LPSTR lpszPath, int idListBox);

BOOL DlgDirSelectComboBox(HWND hwndDlg, LPSTR lpszPath, int idComboBox);

136.2 Description

The *DlgDirSelect()* and *DlgDirSelectComboBox()* functions return information on files selected from controls filled with the *DlgDirList()* and *DlgDirListComboBox()* functions. The *DlgDirSelect()* function retrieves path information from a list box control. The *hwndDlg* parameter contains the window handle of the dialog with the control to be queried. The *lpszPath* parameter contains a pointer to a buffer to be filled with the path or file selected in the list box. The *idListBox* parameter contains the dialog-control identifier of the list box.

DlgDirSelectComboBox() is identical to the *DlgDirSelect()* function, except that it retrieves information from a combobox rather than a list box. Its *idComboBox* parameter contains the dialog-control identifier of the combobox to be queried.

136.3 Returns

The *DlgDirSelect()* and *DlgDirSelectComboBox()* functions return TRUE if successful. Otherwise, they return FALSE.

136.4 Errors

None.

136.5 Cross-References

DlgDirList(), DlgDirListComboBox()

137 EndDialog

137.1 Synopsis

void EndDialog(HWND hwndDlg, int nResult);

137.2 Description

The *EndDialog()* function destroys a modal dialog box and causes the *DialogBox()* function that created it to return. The *hwndDlg* parameter contains the window handle of the dialog box to be destroyed. The *nResult* parameter contains the value to be returned from *DialogBox()*.

If *EndDialog()* is used on a modeless dialog, the dialog will only be hidden. The *DestroyWindow()* function should be used to destroy a modeless dialog.

137.3 Returns

None.

137.4 Errors

None.

137.5 Cross-References

DialogBox(), DestroyWindow()

---

138 GetDialogBaseUnits

138.1 Synopsis

DWORD GetDialogBaseUnits(void);

138.2 Description

GetDialogBaseUnits() returns information on the current dialog-coordinate units. It takes no parameters.

138.3 Returns

The return value is a 32-bit DWORD value containing the size of the standard dialog unit. The high 16 bits of this value contain eight times the pixel height of a vertical dialog unit. The low 16 bits of the DWORD value contain four times the pixel width of a horizontal dialog unit.

138.4 Errors

None.

138.5 Cross-References

None.

---

139 GetDlgCtrlID

139.1 Synopsis

int GetDlgCtrlID(HWND hwndCtrl);

139.2 Description

This function returns the dialog-control identifier of a control in a dialog box or of any child window with an identifier. The *hwndCtrl* parameter contains the window handle whose ID is to be returned.

139.3 Returns

If successful, this function returns the dialog-control identifier of the specified control or child window.

This function returns zero if an error occurs. If a top-level window's handle is passed to *GetDlgCtrlID()*, the return value is non-zero, but invalid.

139.4 Errors

None.

139.5 Cross-References

None.

---

140 GetDlgItem

140.1 Synopsis

HWND GetDlgItem(HWND hwndDlg, int idControl);

140.2 Description

The *GetDlgItem()* function returns the window handle of a control in a dialog box, or of another child window. The *hwndDlg* parameter contains the window handle of the dialog box or the parent window. The *idControl* parameter contains the dialog-control identifier of the control to be queried or the child-window identifier of another child window.

140.3 Returns

If successful, this function returns the window handle of the specified child window or control. This function returns zero if an error occurs.

140.4 Errors

None.

140.5 Cross-References

None.

---

141 GetDlgItemInt, SetDlgItemInt

141.1 Synopsis

UINT GetDlgItemInt(HWND hwndDlg, int idControl, BOOL *pTranslated, BOOL bSigned);

void SetDlgItemInt(HWND hwndDlg, int idControl, UINT uValue, BOOL bSigned);

141.2 Description

The *GetDlgItemInt()* function returns the integer value stored in the text of a control. The *hwndDlg* parameter contains the window handle of the parent window of the control, typically a dialog box. The *idControl* parameter contains the dialog-control identifier of the control to be queried. The *pTranslated* parameter contains a pointer to a boolean value indicating whether a valid value was returned or not. If this parameter is NULL, this function returns no error information. The *bSigned* parameter indicates whether a signed value should be returned. If this parameter is non-zero, the return value of *GetDlgItemInt()* needs to be cast to a signed integer.

SetDlgItemInt() sets the text of a control to contain an integer value, such as the string "123" from the integer 123. The *hwndDlg* and *idControl* parameters are identical to the similarly named parameters of the *GetDlgItemInt()* function. The *uValue* parameter contains the value to be stored in the control. The *bSigned* parameter indicates whether the *uValue* parameter contains a signed or unsigned integer. If this parameter is non-zero, the signed value must be cast to a UINT before being passed to *SetDlgItemInt()*.

141.3 Returns

The *GetDlgItemInt()* function returns the value represented by the text in the specified control, if successful. If it fails, the function returns zero.

If the text in the specified control does not contain a valid number, the *GetDlgItemInt()* function returns zero. If a pointer was passed in the *pTranslated* parameter, the Boolean variable it points to is set to zero. This value is set to a non-zero value if the function is successful.

The *SetDlgItemInt()* function does not return a value.

141.4 Errors

None.

141.5 Cross-References

None.

---

142 GetDlgItemText, SetDlgItemText

142.1 Synopsis

int GetDlgItemText(HWND hwndDlg, int idControl, LPSTR pszText, int nMax);

void SetDlgItemText(HWND hwndDlg, int idControl, LPSTR pszText);

142.2 Description

The *GetDlgItemText()* function retrieves text from a control in a dialog box or another child window. The *hwndDlg* parameter contains the window handle of the dialog box containing the control or the parent window of the window being queried. The *idControl* parameter contains the dialog-item identifier of the control being queried. The *pszText* parameter contains a pointer to a buffer to receive the text. The size of this buffer is passed as the *nMax* parameter.

The *SetDlgItemText()* function sets the text of a control in a dialog box or another child window. The *hwndDlg* and *idControl* parameters are identical to the corresponding parameters in *GetDlgItemText()*. The *pszText* parameter contains a pointer to a null-terminated string containing the new text of the specified control.

142.3 Returns

The *GetDlgItemText()* function returns the number of characters copied to the text buffer. This can be any positive value up to the *nMax* parameter, or zero if the function fails. This function returns zero if an error occurs.

The *SetDlgItemText()* function does not return a value.

142.4 Errors

None.

142.5 Cross-References

None.

143 GetNextDlgGroupItem, GetNextDlgTabItem

143.1 Synopsis

HWND GetNextDlgGroupItem(HWND hwndDlg, HWND hwndCurrent, BOOL bPrev);

HWND GetNextDlgTabItem(HWND hwndDlg, HWND hwndCurrent, BOOL bPrev);

143.2 Description

These functions return the next dialog control which is either in the current group of dialog controls, returned by *GetNextDlgGroupItem()*, or the next control that is a tab-stop, returned by *GetNextDlgTabItem()*. The *hwndDlg* parameter contains the window handle of the dialog box containing the controls. The *hwndCurrent* parameter contains the window handle of the current control. The *bPrev* parameter indicates in which direction the search for a suitable item will go. If this parameter is zero, these functions search forward for the next group or tab-stop item. If it is non-zero, they search backward for the previous item.

A dialog control starts a group of controls if it has the WS_GROUP style. The group continues until the last control not having the WS_GROUP style is found. A control is a tab-stop if it has the WS_TABSTOP style.

143.3 Returns

Both functions return the handle of the next or previous control having the desired attribute, if successful.

Both functions return zero if an error occurs.

143.4 Errors

None.

143.5 Cross-References

None.

144 IsDialogMessage

144.1 Synopsis

BOOL IsDialogMessage(HWND hwndDlg, MSG *pMsg);

144.2 Description

The *IsDialogMessage()* function determines whether a message is intended for a modeless dialog box and dispatches it, if it is. The *hwndDlg* parameter contains the window handle of the modeless dialog box. The *pMsg* parameter contains a pointer to a **MSG** structure containing the message. This function is normally used in a message loop. If it returns zero, indicating that the message was not intended for the dialog, the application should use the *DispatchMessage()* function to dispatch the message.

144.3 Returns

This function returns TRUE if the message was for the dialog box indicated. It returns FALSE if it was not.

144.4 Errors

None.

144.5 Cross-References

DispatchMessage()

---

145 IsDlgButtonChecked

145.1 Synopsis

UINT IsDlgButtonChecked(HWND hwndDlg, int idControl);

145.2 Description

This function determines whether a button control, such as a check box or radio button, is checked. The *hwndDlg* parameter contains the window handle of the dialog box or parent window of the control. The *idControl* parameter contains the dialog-control identifier of the button to be queried.

145.3 Returns

This function returns zero if the button is not checked, one if it is checked, or two if the button is a three-state button and is in the "grayed" state.

IsDlgButtonChecked() returns -1 if the specified control is not a button control that can be checked.

145.4 Errors

None.

145.5 Cross-References

None.

---

146 MapDialogRect

146.1 Synopsis

void MapDialogRect(HWND hwndDlg, RECT *pRect);

146.2 Description

The *MapDialogRect()* function converts dialog units to screen units. The *hwndDialog* parameter contains the window handle of the dialog box for which coordinates are to be converted. The *pRect* parameter contains a pointer to a **RECT** structure containing the coordinates. These coordinates are in dialog units when the function is called. When the function returns, the coordinates are in screen units (pixels).

146.3 Returns

None.

146.4 Errors

None.

146.5 Cross-References

None.

---

147 SendDlgItemMessage

147.1 Synopsis

LRESULT SendDlgItemMessage(HWND hwndDlg, int idControl, UINT message, WPARAM wParam,

LPARAM lParam);

147.2 Description

This function sends a message to a control in a dialog box, or another child window. This function can be used in place of the *SendMessage()* function when the window handle of the control is not known. The *hwndDlg* parameter contains the window handle of the dialog containing the target of the message. The *idControl* parameter contains the dialog-control identifier of the target control or child window. The message parameter contains the message to be sent. The *wParam* and *lParam* parameters contain additional information, depending on the message.

The *SendDlgItemMessage()* function does not return until the control has processed the message. In contrast, the *PostMessage()* function posts the message in the application's message queue and returns immediately.

147.3 Returns

This function returns a 32-bit value. Its precise meaning depends on the message sent.

147.4 Errors

None.

147.5 Cross-References

SendMessage(), PostMessage()

---

148 DlgDirSelectEx

148.1 Synopsis

BOOL DlgDirSelectEx (HWND hwndDlg, LPSTR lpszBuffPath, int nbPath, int ListBoxId);

148.2 Description

The *DlgDirSelectEx()* function fetches the current selection from a list box, which has already been filled by the *DlgDirList()* function. The selection should be a drive letter, a filename, or a directory name. The *hwndDlg* parameter, identifies the handle of the dialog box that contains the list box. The *lpszBuffPath* parameter, contains a pointer to the buffer that will contain the selected path or filename. The *nbPath* parameter, identifies the length in bytes of the filename or directory that is contained in the parameter *lpszBuffPath.* The *ListBoxId* parameter, identifies the list box id in the given dialog box.

DlgDirSelectEx() removes the square brackets which enclose the selected directory name or the hyphens, in case the selection is a drive letter, so the selection can be inserted into a new path or filename. If nothing has been selected in the combo box, the contents of the *lpszBuffPath* parameter remains unchanged. This function does not allow more than one selection to be returned. This function also sends LB_GETCURSEL and LB_GETTEXT messages to the list box. It must be noted here that the list box cannot be a multiple selection list box. If it is, this function returns a non-zero value and the *lpszBuffPath* parameter remains unchanged.

148.3 Returns

DlgDirSelectEx() returns a non-zero value if the current selection from the list box is a directory name. Otherwise, it returns zero.

148.4 Errors

None.

148.5 Cross-References

DlgDirList(), DlgDirListComboBox(), DlgDirSelect(), DlgDirSelectComboBox(), LB_GETCURSEL, LB_GETTEXT

---

149 DialogProc

149.1 Synopsis

BOOL CALLBACK DialogProc(HWND hndlDlg, UINT msg, WPARAM wParam, LPARAM lParam);

149.2 Description

DialogProc() is a callback function, defined by the application, that processes messages that are sent to a modeless dialog box. The *hndlDlg* parameter identifies the dialog box handle. The *msg* parameter identifies the message. The *wParam* parameter identifies the 16 bits of additional message-dependent information. The *lParam* parameter identifies 32 bits of additional message-dependent information.

This function is used only when the dialog box class which is the default class is used for the dialog box. It must be noted here that this function should never call the *DefWindowProc()* function to process unwanted messages. Rather this should be done internally by the dialog box window procedure.

Since the *DialogProc()* function is a placeholder name for the library-defined function name, the EXPORTS statement in the library's module-definition file should have the actual name to be exported.

149.3 Returns

The function *DialogProc()* returns a TRUE value only if the message other than WM_INITDIALOG message is processed. Otherwise, it returns FALSE. If the WM_INITDIALOG message is processed by the function, the return value is FALSE, if the *SetFocus()* function is called in order to set focus to one of the controls in the dialog box. Otherwise, the focus is set to the first control in the dialog box and the return value is TRUE.

149.4 Errors

None.

149.5 Cross-References

CreateDialog(), CreateDialogIndirect(), CreateDialogIndirectParam(), CreateDialogParam(), DefWindowProc(), SetFocus(), WM_INITDIALOG

---

150 BeginPaint

150.1 Synopsis

HDC BeginPaint(HWND hwnd, PAINTSTRUCT *lpps);

150.2 Description

The *BeginPaint()* function prepares the windows identified by *hwnd* for painting and fills the *lpps* parameter with the necessary information to point to the appropriate **PAINTSTRUCT** structure.

150.3 Returns

BeginPaint(), if successful, returns the value of the device-context for the window.

150.4 Errors

None.

150.5 Cross-References

None.

---

151 EndPaint

151.1 Synopsis

void EndPaint(HWND hwnd, PAINTSTRUCT *lpps);

151.2 Description

EndPaint() is called at the end of painting to the window specified by *hwnd*. There should be a call to *EndPaint()* for each call made to *BeginPaint()*. The *lpps* parameter is a pointer to the **PAINTSTRUCT** structure, which was filled by *BeginPaint()*.

151.3 Returns

None.

151.4 Errors

None.

151.5 Cross-References

BeginPaint()

---

152 ExcludeUpdateRgn

152.1 Synopsis

int ExcludeUpdateRgn(HDC hdc, HWND hwnd);

152.2 Description

ExcludeUpdateRgn() prevents a region specified by *hdc* from being updated when the *hwnd* window is redrawn.

152.3 Returns

ExcludeUpdateRgn() returns SIMPLEREGION if there are no overlapping borders in the region, COMPLEXREGION if there are overlapping borders in the region, or NULLREGION if the region is empty.

ExcludeUpdateRgn() returns ERROR if no region is created.

152.4 Errors

None.

152.5 Cross-References

None.

---

153 GetUpdateRect

153.1 Synopsis

BOOL GetUpdateRect(HWND hwnd, RECT *lprc, BOOL fErase);

153.2 Description

GetUpdateRect() gets the coordinates of the smallest rectangle that completely encloses a region of the *hwnd* window that is updated. The coordinates are copied into the **RECT** structure pointed to by the *lprc* parameter. The coordinates returned can be in one of two possible forms. One, if the *hwnd* window was originally created with the CS_OWNDC style and the mapping mode is not MM_TEXT, then the function retrieves logical coordinates of the rectangle, otherwise, client coordinates are returned. Two, the *fErase* parameter determines whether to erase the background in the update region.

153.3 Returns

If the update region is not empty, *GetUpdateRect()* returns a TRUE value. Otherwise, the value returned is FALSE.

153.4 Errors

None.

153.5 Cross-References

None.

---

154 GetUpdateRgn

154.1 Synopsis

int GetUpdateRgn(HWND hwnd, HRGN hrgn, BOOL fErase);

154.2 Description

The *GetUpdateRgn()* function gets the coordinates of the update region of a window specified by *hwnd*. The coordinates returned are relative to the upper-left corner of the *hwnd* window. The *hrgn* parameter identifies the update region whose coordinates are to be returned. The *fErase* parameter determines whether or not to erase the background. If it is set to FALSE, no drawing is done.

154.3 Returns

GetUpdateRect() returns SIMPLEREGION if there are no overlapping borders in the region, COMPLEXREGION if there are overlapping borders in the region, or NULLREGION if the region is empty.

GetUpdateRect() returns ERROR if the function is not successful.

154.4 Errors

None.

154.5 Cross-References

GetUpdateRect()

---

155 InvertRect

155.1 Synopsis

typedef struct tagRECT {

int left;

int top;

int right;

int bottom;

} RECT;

void InvertRect(HDC hdc, const RECT *RectPtr);

155.2 Description

InvertRect() inverts the pixels in a rectangular area of the device-context by performing a logical NOT operation on the each of the pixels' bits. The *hdc* parameter is a handle to the device-context. The *RectPtr* parameter is a pointer to a **RECT** structure that contains the logical coordinates of the rectangular area to invert.

If the device-context is for a monochrome device, black colored pixels inside of the rectangular area will be turned into white colored pixels and vice-versa. If the device-context is for a color device, the way in which pixels inside of the rectangular area are changed depends largely on how the device's colors are generated.

InvertRect() checks the values of the elements in the **RECT** structure pointed to by the *RectPtr* parameter. If one of the following conditions is met, the *InvertRect()* function will not invert the pixels in the rectangular area:

> The value of the **RECT** structure's right element is less than the value of the **RECT** structure's left element.
>
> The value of the **RECT** structure's bottom element is less than the value of the **RECT** structure's top element.

155.3 Returns

None

155.4 Errors

None.

155.5 Cross-References

RECT

---

156 UpdateWindow

156.1 Synopsis

void UpdateWindow(HWND hwnd);

156.2 Description

UpdateWindow() updates a non-empty window identified by *hwnd,* by sending the WM_PAINT message to it. The WM_PAINT message is sent directly to the window, bypassing the message queue.

156.3 Returns

None.

156.4 Errors

None.

156.5 Cross-References

None.

---

157 ValidateRgn, InValidateRgn

157.1 Synopsis

void ValidateRgn(HWND hWnd, HRGN hRgn);

void InvalidateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);

157.2 Description

The *ValidateRgn()* function removes a region of a window's client area from the window's current update region. The *hWnd* parameter is the handle of the window whose client area is being validated. The *hRgn* parameter is a handle to a region that contains the client area coordinates to validate. If the value of the *hRgn* parameter is NULL, all of the client area are validated.

The *InvalidateRgn()* function adds a region of a window's client area to the window's current update region. The *hWnd* parameter is the handle of the window whose client area is being invalidated. The *hRgn* parameter is a handle to a region that contains the client area coordinates to invalidate. If the value of the *hRgn* parameter is NULL, all of the client area are invalidated. The *bErase* parameter defines whether all of the update region's background (not just the region being invalidated) should be erased when it is processed. The *bErase* parameter can be one of the following values:

TRUE	All of the update region's background should be erased when it is processed.
FALSE	All of the update region's background should not be erased when it is processed.

157.3 Returns

None

157.4 Errors

None.

157.5 Cross-References

BeginPaint(), WM_PAINT, *ValidateRect(), InValidateRect()*

---

158 RedrawWindow

158.1 Synopsis

BOOL RedrawWindow(HWND hndlwnd, const RECT *lpUpdateRect, HRGN hndlUpdateRgn,

UINT RedrawFlag);

158.2 Description

The *RedrawWindow()* function updates the specified region or rectangle in the client area of the given window. The *hndlwnd* parameter identifies the handle of the window. The *lpUpdateRect* parameter contains a pointer to the **RECT** structure identifying the coordinates of the rectangle to be updated. The *hndlUpdateRgn* parameter contains the handle for the region to be updated. If both of the above parameters are NULL, the entire window's client area is updated. The *RedrawFlag* parameter contains one or a combination of the redraw flags and can have one of the following values:

RDW_ERASE	This value 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	This value causes any part of the non-client area of the window that intersects the update region to receive a WM_NCPAINT message. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_FRAME has no effect. The WM_NCPAINT message is typically not sent during the execution of the *RedrawWindow()* function unless either RDW_UPDATENOW or RDW_ERASENOW is specified.
RDW_INTERNALPAINT	This value causes a WM_PAINT message to be posted to the window, regardless of whether the window contains an invalid region.
DW_INVALIDATE	This value 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:

RDW_NOERASE	This flag suppresses any pending WM_ERASEBKGND messages.
RDW_NOFRAME	This flag suppresses any pending WM_NCPAINT messages. This flag must be used with RDW_VALIDATE and is typically used with NOCHILDREN. Use this option with care, as it could cause problems painting parts of a window properly.
RDW_NOINTERNALPAINT	This flag suppresses any pending internal WM_PAINT messages; this flag does not affect WM_PAINT messages resulting from invalid areas.
RDW_VALIDATE	This flag validates *pUpdateRect* or *hndlUpdateRgn* (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 repainting. No painting is performed by the *RedrawWindow()* function unless one of these bits is specified.

RDW_ERASENOW	This flag causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive WM_NCPAINT and WM_ERASEBKGND messages, if necessary, before the function returns. WM_PAINT messages are deferred.
RDW_UPDATENOW	This flag 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.

If the specified window has the style WS_CLIPCHILDREN set, the child windows of the specified window are not affected by the *RedrawWindow().* However, windows that do not have this style set will be recursively validated or invalidated until a window that has the WS_CLIPCHILDREN style found. The following flags control which windows are affected by the *RedrawWindow()* function:

RDW_ALLCHILDREN	This flag includes child windows, if any, in the repainting operation.
RDW_NOCHILDREN	This flag excludes child windows, if any, from the repainting operation.

If this function is used to invalidate a part of the desktop window, the desktop window will not receive the WM_PAINT message. This should be done by the application, which uses the RDW_ERASE flag to generate the WM_ERASEBKGND message.

158.3 Returns

If the *RedrawWindow()* is successful, it returns TRUE. Otherwise, it returns FALSE.

158.4 Errors

None.

158.5 Cross-References

GetUpdateRect(), GetUpdateRgn(), InvalidateRect(), InvalidateRgn(), UpdateWindow(), WM_ERASEBKGND, WM_PAINT

---

159 LockWindowUpdate

159.1 Synopsis

BOOL LockWindowUpdate(HWND hLockWnd);

159.2 Description

The *LockWindowUpdate()* function enables or disables a drawing in the window specified by the *hndlwndLock* parameter. A window that has been locked cannot be moved. Only one window can be locked at a time. The *hLockWnd* parameter identifies the handle of the window that locks or disables the drawing. If the value of this parameter is NULL, drawing is permitted in the locked window. If an application that has locked a parent or child window, makes a call to *GetDC(), GetDCEx(),* or *BeginPaint()* the returned device-context will specify a visible region that is empty. This state can be reversed by making a call to the *LockWindowUpdate()* function with the *hLockWnd* parameter, set to NULL.

Note: While the window updates are locked, the system will keep track of drawing operations to the device-contexts associated with the locked window on the bounding rectangle.

If the drawing is reenabled, the bounding rectangle is invalidated in both the locked window and its child window. This forces a WM_PAINT message that updates the screen area. However, if no drawing has occurred during the time the window remained locked, none of the areas are invalidated. This function does not clear the WS_VISIBLE style bit or make the given window invisible.

159.3 Returns

If this function is successful, the return value is TRUE. It returns FALSE if the function fails or if this function has been used to lock another window.

159.4 Errors

None.

159.5 Cross-References

None.

---

160 GetMapMode, SetMapMode

160.1 Synopsis

int GetMapMode(HDC hdc);

int SetMapMode(HDC hdc, int nMapMode);

160.2 Description

The *GetMapMode()* function retrieves the current mapping mode for the specified device-context, while *SetMapMode()* sets its value. Mapping mode defines how to convert logical coordinates into the appropriate device coordinates. Parameter *nMapMode*, in the *SetMapMode()* function can be any of the following values: MM_ANISOTROPIC, MM_HIENGLISH, MM_HIMETRIC, MM_ISOTROPIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, or MM_TWIPS.

160.3 Returns

GetMapMode() returns the mapping mode value if it is successful. This can one of the following values: MM_ANISOTROPIC, MM_HIENGLISH, MM_HIMETRIC, MM_ISOTROPIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT or MM_TWIPS.

SetMapMode() returns the previous mapping mode.

160.4 Errors

None.

160.5 Cross-References

SetViewPortExt(), SetWindowExt()

161 GetPolyFillMode, SetPolyFillMode

161.1 Synopsis

int GetPolyFillMode(HDC hdc);

int SetPolyFillMode(HDC hdc, int nMode);

161.2 Description

The *GetPolyFillMode()* function retrieves the current polygon filling mode for the specified device-context. *SetPolyFillMode()* sets its value. Parameter *nMode* in *SetPolyFillMode()* can be either ALTERNATE or WINDING. For ALTERNATE polygon-filling mode, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. When the mode is WINDING, each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction.

161.3 Returns

GetPolyFillMode() returns current polygon filling mode, which can be either ALTERNATE or WINDING. *SetPolyFillMode()* returns previous polygon filling mode, if it is successful. Otherwise, it returns zero.

161.4 Errors

None.

161.5 Cross-References

PolyPolygon()

162 GetROP2, SetROP2

162.1 Synopsis

int GetROP2(HDC hdc);int SetROP2(HDC hdc, int nDrawMode);

162.2 Description

The *GetROP2()* function retrieves the current drawing mode for the specified device-context, while the *SetROP2()* function sets that mode. The drawing mode defines, how the colors of new objects are combined with the color already on the screen. It applies for raster devices only and does not make sense for vector devices. The *nDrawMode* parameter in the *SetROP2()* function can be one of the following values:

R2_BLACK	This value specifies that the pixel is always black.
R2_WHITE	This value specifies that the pixel is always white.
R2_NOP	This value specifies that the pixel remains unchanged.
R2_NOT	This value specifies that the pixel is the inverse of the existing screen color.
R2_COPYPENT	This value specifies that the pixel is the color of currently selected pen.
R2_NOTCOPYPEN	This value specifies that the pixel is the inverse color of currently selected pen.
R2_MERGEPENNOT	This value specifies that the ixel is a combination of the pen color and the inverse of the screen color.
R2_MASKPENNOT	This value specifies that the pixel is a combination of the colors common to both the pen and the inverse of the screen.

R2_MERGENOTPEN	This value specifies that the pixel is a combination of the screen color and the inverse of the pen color.
R2_MASKNOTPEN	This value specifies that the pixel is a combination of the colors common to both the screen and the inverse of the pen.
R2_MERGEPEN	This value specifies that the pixel is a combination of the pen color and the screen color.
R2_NOTMERGEPEN	This value specifies that the pixel is the inverse of the R2_MERGEPEN color.
R2_MASKPEN	This value specifies that the pixel is a combination of the colors common to both the pen and the screen.
R2_NOTMASKPEN	This value specifies that the pixel is the inverse of the R2_MASKPEN color.
R2_XORPEN	This value specifies that the pixel is a combination of the colors that are in the pen and in the screen, but not in both.
R2_NOTXORPEN	This value specifies that the pixel is the inverse of the R2_XORPEN color.

162.3 Returns

GetROP2() returns the current drawing mode, if it is successful. The *GetROP2()* function returns the previous drawing mode.

162.4 Errors

None.

162.5 Cross-References

GetDeviceCaps()

163 GetBkColor, SetBkColor

163.1 Synopsis

COLORREF GetBkColor(HDC hdc);

COLORREF GetBkColor(HDC hdc, COLORREF clrref);

163.2 Description

The *GetBkColor()* function returns the current background color for the given device-context, while function *SetBkColor()* sets the color. Parameter *clrref* in the *SetBkColor()* function specifies the new color. If the output device cannot display the new color, the systems selects the nearest physical color value.

163.3 Returns

The *GetBkColor()* function returns the RGB color value, if it is successful. The *SetBkColor()* function returns the RGB value of the previous background color if it is successful, or 0x80000000 if it fails.

163.4 Errors

None.

163.5 Cross-References

GetBkMode(), SetBkMode()

164 GetBkMode, SetBkMode

164.1 Synopsis

int GetBkMode(HDC hdc);

int SetBkMode(HDC hdc, int nBkMode);

164.2 Description

The *GetBkMode()* function the background mode for the device-context. Background mode defines how the system handles existing background colors when drawing text, hatched brushes, or any non-solid pen. *SetBkMode()* sets the background mode for the device-context. The *nBkMode* parameter can be one of the following values:

OPAQUE	Before drawing text, hatched brush or non-solid line, the background is filled with the current background color.
TRANSPARENT	Background is not changed before drawing.

164.3 Returns

GetBkColor() returns the current background mode that can be either OPAQUE or TRANSPARENT. *SetBkColor()* returns the previous background mode, if it is successful.

164.4 Errors

None.

164.5 Cross-References

GetBkColor(), SetBkColor()

165 GetBoundsRect, SetBoundsRect

165.1 Synopsis

UINT GetBoundsRect(HDC hdc, RECT *lprcBounds, UINT flags);

UINT SetBoundsRect(HDC hdc, RECT *lprcBounds, UINT flags);

165.2 Description

The *GetBoundsRect()* function returns the accumulated bounding rectangle in *lprcBounds* for the given device-context. *SetBoundsRect()* controls the bounding rectangle accumulation. If the *flags* parameter in *GetBoundsRect()* function is set to DCB_RESET, the bounding rectangle is cleared after the function returns. Otherwise, it should be zero. The *flags* parameter of the *SetBoundsRect()* function may be any combination of the following values:

DCB_ACCUMULATE	This value adds the *lprcBounds* rectangle to current bounding rectangle using union operation.
DCB_DISABLE	This value disables the bounding rectangle accumulation (default setting).
DCB_ENABLE	This value enables bounding rectangle accumulation.

165.3 Returns

The *GetBoundsRect()* function returns DCB_SET if the bounding rectangle is not empty, otherwise, it returns DCB_RESET. The *SetBoundsRect()* function returns the current state of bounding rectangle accumulations, if it is successful.

165.4 Errors

None.

165.5 Cross-References

BeginPaint(), EndPaint()

166 GetCurrentPosition, GetCurrentPositionEx

166.1 Synopsis

DWORD GetCurrentPosition(HDC hdc);

BOOL GetCurrentPositionEx(HDC hdc, POINT *lpPoint);

166.2 Description

The *GetCurrentPosition()* and *GetCurrentPositionEx()* functions retrieve the logical coordinates of the current position. *GetCurrentPosition()* returns the coordinates as the function value, while *GetCurrentPositionEx()* stores the coordinates in the **POINT** structure, specified by the *lpPoint* parameter.

166.3 Returns

GetCurrentPosition() returns the logical x-coordinate in low-order word and the logical y-coordinate in high-order word, if it is successful. *GetCurrentPositionEx()* returns non-zero if it is successful and zero if it fails.

166.4 Errors

None.

166.5 Cross-References

CreateDC(), DeleteDC()

---

167 InvalidateRect, ValidateRect

167.1 Synopsis

void InvalidateRect(HWND hWnd, const RECT *lprc, BOOL fErase);

void ValidateRect(HWND hWnd, const RECT *lprc);

167.2 Description

The *InvalidateRect()* function adds the rectangle, specified by lprc parameter, to the window's update region. The update region defines which part of the window's client area must be redrawn. The WM_PAINT message is sent to the window when the update region is not empty and there are no other messages waiting for that window. If the *fErase* parameter in the *InvalidateRect()* function is TRUE, the background of entire update region is erased when the *BeginPaint()* function is called. Otherwise, the background remains unchanged.

The *ValidateRect()* function validates the rectangular part of the client area, specified by lprc parameter by removing that rectangle from the window's update region. The *BeginPaint()* function automatically validates the entire client area.

If the *lprc* parameter is equal NULL, the entire client area is processed by both the *InvalidateRect()* and *ValidateRect()* functions.

167.3 Returns

None.

167.4 Errors

None.

167.5 Cross-References

CreateDC(), DeleteDC()

Section 3 - Graphics Subsystem

168 CreateCompatibleDC

168.1 Synopsis

HDC CreateCompatibleDC(HDC hdc);

168.2 Description

The *CreateCompatibleDC()* function creates a new memory device-context that is compatible with the device-context specified in the *hdc* parameter. If zero is passed in the hdc parameter, the new device-context will be compatible with the screen. The memory device-context can then be used for off-screen drawing, or for preparing images for copying to the screen. This can be accomplished by selecting bitmaps into the image or drawing into the image.

The compatible device-context can be created only for a device that supports raster operations. This can be determined by calling the *GetDeviceCaps()* function with the RC_BITBLT index.

The *DeleteDC()* function should be used to dispose of the memory device-context.

168.3 Returns

The function returns the handle of the created memory device-context, if it is successful. Otherwise, it returns zero.

168.4 Errors

None.

168.5 Cross-References

DeleteDC()

---

169 CreateDC, CreateIC

169.1 Synopsis

HDC CreateDC(LPCSTR lpszDriver, LPCSTR lpszDevice, LPCSTR lpszOutput, const void *lpvInitData);

HDC CreateIC(LPCSTR lpszDriver, LPCSTR lpszDevice, LPCSTR lpszOutput, const void *lpvInitData);

169.2 Description

The *CreateDC()* function creates a device-context for the given device; the *CreateIC()* function creates an information context for the device. The information context provides a way to retrieve device attributes, but does not create the underlying driver structures necessary to draw into it. Its creation is much faster than a call to *CreateDC()* for the same device.

The *lpszDriver* parameter is a pointer to a null-terminated name of the device driver without the filename extension.

The *lpszDevice* parameter points to the name of the device, for the drivers that can support more than one device. This parameter can be NULL if the driver name is sufficient.

The *lpszOutput* parameter is a pointer to the device name of the output port or a file name, if the output is redirected to a file.

The *lpvInitData* parameter is a pointer to the **DEVMODE** structure that contains the device-specific information for the driver, or NULL if the driver is expected to use the default initialization data. The **DEVMODE** structure should have been filled in by the previous call to the *ExtDeviceMode()* function to the driver. The structure is defined in <print.h>.

DeleteDC() should be used to dispose of both information context and device-context created by these functions.

169.3 Returns

The function returns the handle of the created device or information context, if it is successful. Otherwise, it returns zero.

169.4 Errors

None.

169.5 Cross-References

DeleteDC(), ExtDeviceMode()

---

170 GetDC, GetWindowDC, GetDCEx

170.1 Synopsis

```
HDC GetDC(HWND hWnd);

HDC GetWindowDC(HWND hWnd);

HDC GetDCEx(HWND hWnd, HRGN hrgnClip, DWORD dwFlags);
```

170.2 Description

The *GetDC()* function obtains a handle of the device-context (DC) for the client area of the given window. The origin of the DC is set to the upper-left corner of the window's client area, and the DC is clipped by the visible part of the client area. It is equivalent to the following call:

```
GetDCEx(hWnd, (HRGN)0, DCX_DEFAULTCLIP);
```

The *GetWindowDC()* function retrieves the handle of the device-context for the given window, including both the client and non-client areas. The origin of the window DC is set to the upper-left corner of the window frame, and it will be clipped by the visible area of the window, client and non-client areas included. Using the window DC allows the application to draw frames, captions system scroll bars, menu bars, and so on, into window's non-client area. The call to *GetWindowDC()* is equivalent to the following call:

```
GetDCEx(hWnd, (HRGN)0, DCX_DEFAULTCLIP|DCX_WINDOW|DCX_CACHE
```

The *GetDCEx()* function obtains the handle of the device-context of the given window. The clipping and origin of the DC will depend on the flags passed to the function. The *hWnd* parameter specifies the handle of the device-context for the window. If this parameter is zero, the device-context for the desktop window will be obtained. The *hrgnClip* parameter of *GetDCEx()* is a handle of the clipping region that might be used depending on the flags passes in *dwFlags* parameter.

The *dwFlags* parameter specifies a combination of the following options:

DCX_CACHE	This option specifies that the DC should be taken from the DC cache, overriding the CS_OWNDC and CS_CLASSDC class styles.
DCX_CLIPCHILDREN	This option specifies that the regions of the window that are covered by the window's children will be excluded from the DC's visible region.
DCX_CLIPSIBLINGS	This option specifies that the regions of the window covered by its siblings that are above it in the Z-order will be excluded from the DC's visible region.
DCX_EXCLUDERGN	This option specifies that the region specified in *hrgnClip* parameter will be excluded from the DC's visible region.
DCX_INTERSECTRGN	This option specifies that the region specified in *hrgnClip* parameter will be intersected with the visible region of the DC.
DCX_LOCKWINDOWUPDATE	This option overrides the lock on drawing into the window set by a previous call to the *LockWindowUpdate()* for the same window, so that drawing into this DC is allowed.
DCX_PARENTCLIP	This option specifies that the DC is clipped by the parent's rectangle, and the DC origin is set to the upper-left corner of the parent's rectangle, thus allowing the window to draw outside its own area. WS_CLIPCHILDREN style on parent and *hWnd* windows are ignored. If this option is used in combination with the DCX_WINDOW flag, the

	parent's window rectangle is being used; otherwise, the parent's client rectangle is used.
DCX_WINDOW	This option specifies that the DC is clipped by the window rectangle, not the client rectangle, of the window; the WS_CLIPCHILDREN style of the window is ignored.

ReleaseDC() should be used to release the device-context after drawing. It should be called with the same *hWnd* value as the call that obtained the DC being released.

If the *hWnd* class belongs to the same class as the CS_CLASSDC style, all windows of this class get the same device-context, unless this style is overridden by the DCX_CACHE, DCX_WINDOW, or DCX_PARENTCLIP flags.

If the window class has the CS_OWNDC style, the window gets its own DC on creation and it is only released when the *DestroyWindow()* function is called. The CS_PARENTDC class style forces the DC to be set to the parent window, but the origin of the DC will be set to the upper-left corner of the window, not the parent.

Unless the window has either the CS_OWNDC or CS_CLASSDC style, the device-context in the case of a call to *GetDC()* or *GetWindowDC(),* will be taken from the DC cache which is limited to five DC's. If *GetDCEx()* is called for such a window, the DCX_CACHE flag should be specified, otherwise, the function call will fail.

The visible region in the device-context is affected by WS_CLIPCHILDREN and WS_CLIPSIBLINGS window styles.

Since the cache size is limited, all device-contexts should be released immediately after use.

170.3 Returns

This function returns the handle of the device-context, if it is successful. Otherwise, it returns zero.

170.4 Errors

None.

170.5 Cross-References

ReleaseDC()

---

171 DeleteDC

171.1 Synopsis

BOOL DeleteDC(HDC hdc);

171.2 Description

The *DeleteDC()* function deletes the device-context specified in the *hdc* parameter. This device-context(DC) is either a memory DC created by a *CreateCompatibleDC()* call or a device-context created for a specific device by a call to *CreateDC()* or *CreateIC()*. It should not be called to dispose of a screen DC obtained by a call to *GetDC(), GetWindowDC(),* or *GetDCEx()*. The *ReleaseDC()* function should be used instead.

All the objects selected by the application into the DC should be selected out of it before the *DeleteDC()* function is called.

171.3 Returns

The function returns TRUE if it is successful. Otherwise, it returns FALSE.

171.4 Errors

None.

171.5 Cross-References

CreateDC(), CreateIC(), CreateCompatibleDC()

172 ReleaseDC

172.1 Synopsis

BOOL ReleaseDC(HWND hWnd, HDC hdc);

172.2 Description

The *ReleaseDC()* function releases the device-context specified in the *hdc* parameter. This device-context should be a screen DC obtained by a previous call to the *GetDC()*, *GetWindowDC()*, or *GetDCEx()* functions. The *hWnd* parameter indicates the handle of the specified window. The DC will not be released if the window belongs to a class that has a CS_OWNDC or CS_CLASSDC style, though the function call will not fail.

172.3 Returns

The function returns TRUE if it is successful. Otherwise, it returns FALSE.

172.4 Errors

None.

172.5 Cross-References

GetDC(), GetWindowDC(), GetDCEx()

173 SaveDC, RestoreDC

173.1 Synopsis

int SaveDC(HDC hdc);

BOOL RestoreDC(HDC hdc, int nSavedDC);

173.2 Description

The *SaveDC()* function saves the current state of the device-context specified in the *hdc* parameter. The state of the DC includes all the objects currently selected into it. These include drawing attributes such as foreground/background colors, background mode, viewport, DC, window origins, mapping mode, clipping region, and so on. The function returns an identifier of the saved device-context. This identifier will be used later to restore the DC. There can be a stack of several saved DC's for a given hdc.

The *RestoreDC()* function restores the state of the device-context specified by the *hdc* parameter using the information associated with the *nSavedDC* parameter. This identifier is either the value previously returned from a *SaveDC()* call or it is -1. The -1 value corresponds to the most recently saved copy of the DC.

If the *RestoreDC()* function restores a previous copy of the DC, all copies of the DC created after this copy will be destroyed, thus freeing all memory associated with these DCs.

173.3 Returns

The *SaveDC()* function returns the identifier of the saved copy of the device-context. It returns zero if it is unsuccessful.

The *RestoreDC()* function returns TRUE if it is successful. It returns FALSE if it is unsuccessful.

173.4 Errors

None.

173.5 Cross-References

None.

174 ResetDC

174.1 Synopsis

HDC ResetDC(HDC hdc, const DEVMODE *lpdm);

174.2 Description

The *ResetDC()* function updates the device-context based on information in the DEVMODE structure, specified by the *lpdm* parameter. Windows applications usually call the *ResetDC()* function in response to the WM_DEVMODECHANGE message. Before calling the *ResetDC()* function, the application must deselect all objects associated with the device-context (this does not apply to stock objects).

174.3 Returns

This function returns the handle of the original device-context. Otherwise, it returns NULL.

174.4 Errors

None.

174.5 Cross-References

DeviceCapabilities(), Escape(), ExtDeviceMode()

---

175 GetDCOrg

175.1 Synopsis

DWORD GetDCOrg(HDC hdc);

175.2 Description

The *GetDCOrg()* function retrieves the coordinates of the final translation origin for the device-context, specified by *hdc* parameter. The origin defines the offset that is used to translate device specific coordinates into window client coordinates. The final translation origin is always relative to the physical screen origin.

175.3 Returns

GetDCOrg() returns the x-coordinate of the origin in the low-order word and the y-coordinate in the high-order word, if it is successful. Both values are in device coordinates.

175.4 Errors

None.

175.5 Cross-References

CreateIC()

---

176 CopyIcon

176.1 Synopsis

HICON CopyIcon(HINSTANCE hndlappinst, HICON hndlicon);

176.2 Description

The *CopyIcon()* function makes a copy of the icon. The *hndlappinst* parameter specifies the instance of the application that copies the icon. The *hndlicon* parameter specifies the icon that is to be copied. The purpose of this function is to provide the application a copy of the icon resource that the application owns. Normally resources are owned by the originating module but can be shared by the application or dynamic link library as long as they are not freed.

If the application or the dynamic-link library needs to use this resource a copy of the resource has to be made. The *CopyIcon()* function performs this for the application or dynamic-link library. The owner of the copied icon resource should call the *DestroyIcon()* function to free the icon resource when it is no longer needed.

176.3 Returns

If the function is successful it returns the handle of the icon that has been copied. Otherwise, the return value is set to NULL.

176.4 Errors

None.

176.5 Cross-References

CopyCursor(), DestroyIcon(), DrawIcon()

---

177 DrawIcon

177.1 Synopsis

BOOL DrawIcon(HDC hdevc, int xcord, int ycord, HICON hndlicon);

177.2 Description

The *DrawIcon()* function draws the icon specified by the hndlicon parameter on the device specified by the *hdevc* parameter. The upper-left hand corner of the icon will be placed at the logical location specified by the parameters *xcord* and *ycord*. The *hdevc* parameter specifies the device-context for the window. The *xcord* parameter identifies the x-coordinate of the upper-left corner of the icon and the *ycord* parameter identifies the y-coordinate of the upper-left corner of the icon. The *hndlicon* parameter specifies the handle of the icon that is drawn.

Note: The application has to first call the function *LoadIcon()* with the mapping mode set to MM_TEXT before the *DrawIcon()* functions can be called.

177.3 Returns

If the function is successful, it returns a TRUE value. Otherwise, it returns FALSE.

177.4 Errors

None.

177.5 Cross-References

GetMapMode(), LoadIcon(), SetMapMode()

---

178 DestroyIcon

178.1 Synopsis

BOOL DestroyIcon(HANDLE hndlicon);

178.2 Description

The *DestroyIcon()* function destroys the icon resource that had been created using the *CreateIcon()* or the *LoadIcon()* functions. This function also frees any memory that is associated with this icon resource. The *hndlicon* parameter specifies the handle of the icon resource that needs to be destroyed.

178.3 Returns

If the function is successful, it returns a TRUE value. Otherwise, it returns FALSE.

178.4 Errors

None.

178.5 Cross-References

CreateCursor(), CreateIcon(), DestroyCursor(), LoadIcon()

---

179 CreateIcon

179.1 Synopsis

HICON WINAPI CreateIcon(HINSTANCE hInst, int nWidth, int nHeight, BYTE bPlanes,

    BYTE bBitsPerPixel, const void *lpvANDbits, const void *lpvXORbits);

179.2 Description

The *CreateIcon()* function creates an icon using the specified parameters. The *hInst* parameter specifies the instance for which the icon will be created.

The *lpvXORbits* parameter is a pointer to an array of bytes that makeup the monochrome or color bitmap XOR mask. The *bBitsPerPixel* parameter specifies the number of bits per pixel that are used in the *lpvXORbits* mask. The *bPlanes* parameter specifies the number of planes used in the *lpvXOR*bits mask.

The *lpvANDbits* is a pointer to an array of bytes that makeup the monochrome AND mask of the icon.

The *nWidth* and *nHeight* parameters determine the dimensions of the icon in pixels and must be supported by the current display driver. The dimensions supported by the current driver may be obtained using the *GetSystemMetrics()* function using the SM_CXICON or SM_CYICON constant.

179.3 Returns

The function returns a handle to the newly created icon, if successful. If the function fails, NULL is returned.

179.4 Errors

None.

179.5 Cross-References

DestroyIcon()

---

180 UnrealizeObject

180.1 Synopsis

BOOL UnrealizeObject(HGDIOBJ hObject);

180.2 Description

The *UnrealizeObject()* function resets a brush or logical palette object. The *hObject* parameter is a handle to the brush or logical palette object that is reset.

If the *hObject* parameter contains a handle to a brush object. The origin of the brush is reset the next time it is selected.

If the *hObject* parameter contains a handle to a logical palette. The palette is set as if it has not been realized previously. The next time an application calls the *RealizePalette()* function for the palette, the system completely remaps the logical palette to the system palette.

180.3 Returns

UnrealizeObject() returns TRUE, if it is successful. Otherwise, it returns FALSE.

180.4 Errors

None.

180.5 Cross-References

RealizePalette()

---

181 PtVisible, RectVisible

181.1 Synopsis

BOOL PtVisible(HDC hdc, int XPos, int YPos);

BOOL RectVisible(HDC hdc, RECT *RectPtr);

181.2 Description

The *PtVisible()* function determines if a point is in a device-context's clipping region. The *hdc* parameter is a handle to a device-context. The *XPos* parameter is the point's logical x-coordinate. The *YPos* parameter is the point's logical y-coordinate.

RectVisible() determines if any part of the rectangle is in a device-context's clipping region. The *hdc* parameter is a handle to a device-context. The *RectPtr* parameter is a pointer to a rectangle containing logical coordinates.

181.3 Returns

PtVisible() returns TRUE if the given point is in the device-context's clipping region. *PtVisible()* returns FALSE if the given point is not in the device-context's clipping region.

RectVisible() returns TRUE if any part of the rectangle is in the device-context's clipping region. *RectVisible()* returns FALSE if no part of the rectangle is in the device-context's clipping region.

181.4 Errors

None.

181.5 Cross-References

None.

---

182 SelectObject

182.1 Synopsis

HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hObject);

182.2 Description

The *SelectObject()* function selects an object into a device-context. The *hdc* parameter is a handle to a device-context. The *hObject* parameter is a handle to the object that should be selected into the device-context. The object must have been created by one of the following functions:

Bitmap object	*CreateBitmap(), CreateBitmapIndirect(),CreateCompatibleBitmap(), CreateDIBitmap()*
Brush object	*CreateBrushIndirect(), CreateDIBPatternBrush(),CreateHatchBrush(), CreatePatternBrush(), CreateSolidBrush()*
Font object	*CreateFont(), CreateFontIndirect()*
Pen object	*CreatePen(), CreatePenIndirect()*
Region object	*CreateEllipticRgn(), CreateEllipticRgnIndirect(),CreatePolygonRgn(), CreateRoundRectRgn(), CreateRectRgn(), CreateRectRgnIndirect()*

A bitmap object can only be selected into a memory device-context. A bitmap object can be selected into only one memory device-context at a time. The format of the bitmap must either be monochrome or be compatible with the given device. If the format of the bitmap is not compatible, the *SelectObject()* function returns an error.

182.3 Returns

In most cases, if the *SelectObject()* function is successful, it returns a handle to the previously selected object. If the *SelectObject()* function is not successful, it returns NULL. Two exceptions to this rule are listed below:

If the *hdc* parameter contains a handle to a metafile device-context, the *SelectObject()* function does not return the handle of the previously selected object.

If the *hObject* parameter is a handle to a region object, *SelectObject()* performs the same operation as the *SelectClipRgn()* function. The *SelectObject()* function's return value will be one of the following:

SIMPLEREGION	The region has no overlapping borders.
COMPLEXREGION	The region has overlapping borders.
NULLREGION	The region is empty.

If an error occurs, the function's return value is ERROR and the previously selected object of the specified type remains selected in the device-context.

182.4 Errors

None.

182.5 Cross-References

CreateBitmap(), CreateBitmapIndirect(), CreateBrushIndirect(), CreateCompatibleBitmap(), CreateDIBitmap(), CreateDIBPatternBrush(), CreateEllipticRgn(), CreateEllipticRgnIndirect(), CreateFont(), CreateFontIndirect(), *CreateHatchBrush(), CreatePatternBrush(), CreatePen(), CreatePenIndirect(), CreatePolygonRgn(), CreateRectRgn(), CreateRectRgnIndirect(), CreateRoundRectRgn(), CreateSolidBrush()*

183 DeleteObject

183.1 Synopsis

BOOL DeleteObject(HGDIOBJ hObject);

183.2 Description

The *DeleteObject()* function frees all memory associated with a pen, brush, font, bitmap, region, or palette object. The *hObject* parameter is a handle to the object that is deleted. An application should not delete an object that is currently selected into a device-context. If a pattern brush is deleted, the bitmap object associated with the pattern brush is not deleted. The bitmap object must be deleted by making another call to the *DeleteObject()* function.

183.3 Returns

If the *DeleteObject()* function is successful, it returns TRUE. If the *DeleteObject()* function is not successful, it returns FALSE.

183.4 Errors

None.

183.5 Cross-References

None.

184 EnumObjects, EnumObjectsProc

184.1 Synopsis

int EnumObjects(HDC hdc, int ObjectType, GOBJENUMPROC EnumObjectsProc, LPARAM lParam);

int CALLBACK EnumObjectsProc(void *ObjectPtr, LPARAM lParam);

184.2 Description

The *EnumObjects()* function enumerates all objects of a given type that are available in a device-context and passes the object information to a user-defined callback function. The *hdc* parameter is a handle to a device-context. The *ObjectType* parameter identifies the type of object to enumerate and can be one of the following values:

OBJ_BRUSH	This value enumerates brush objects.
OBJ_PEN	This value enumerate pen objects.

Each object's information and the value of the *lParam* parameter are passed to the callback function specified in the *EnumObjectsProc()* parameter. The *EnumObjects()* function enumerates the objects until there are no more objects or until the *EnumObjectsProc()* callback function returns a value of FALSE.

An *EnumObjectsProc()* function is a user-defined, exported, enumeration callback function of type GOBJENUMPROC whose address is passed to the *EnumObjects()* function. The *EnumObjects()* function passes a pointer to an object in the *EnumObjectsProc()* function's *ObjectPtr* parameter. The *EnumObjects()* function passes the user-defined value that it originally received when it was called in the *EnumObjectsProc()* function's *lParam* parameter.

184.3 Returns

EnumObjects() returns the last value returned by the *EnumObjectsProc()* callback function.

EnumObjectsProc() returns FALSE to abort the enumeration or TRUE to continue the enumeration.

184.4 Errors

None.

184.5 Cross-References

None.

185 GetObject

185.1 Synopsis

int GetObject(HGDIOBJ hObject, int BufferSize, void *BufferPtr);

185.2 Description

The *GetObject()* function retrieves information about a given object and places the information into the given buffer. The *hObject* parameter is a handle to the object. The *BufferPtr* parameter is a pointer to a buffer to be filled. The *BufferSize* parameter, specifies the number of bytes to copy to the buffer.

If the *hObject* parameter is a handle to a bitmap object, the *GetObject()* function retrieves only the bitmap's width, height and color format.

If the *hObject* parameter is a handle to a logical palette object, the *GetObject()* function retrieves only an integer that specifies the number of entries in the palette.

185.3 Returns

If successful, *GetObject()* returns the number of bytes retrieved. If not successful, *GetObject()* returns zero.

185.4 Errors

None.

185.5 Cross-References

None.

186 GetStockObject

186.1 Synopsis

HGDIOBJ GetStockObject(int ObjectID);

186.2 Description

The *GetStockObject()* function returns a handle to a pen, brush, or font stock object. The *ObjectID* parameter contains the system object's identifier and can be one of the following values:

BLACK_BRUSH	This value specifies a black brush.
DKGRAY_BRUSH*	This value specifies a dark-gray brush.
GRAY_BRUSH*	This value specifies a gray brush.
HOLLOW_BRUSH	This value specifies a hollow brush.
LTGRAY_BRUSH*	This value specifies a light-gray brush.
NULL_BRUSH	This value specifies a null brush.
WHITE_BRUSH	This value specifies a white brush.
BLACK_PEN	This value specifies a black pen.
NULL_PEN	This value specifies a null pen.
WHITE_PEN	This value specifies a white pen.
ANSI_FIXED_FONT	This value specifies a fixed-pitch system font.
ANSI_VAR_FONT	This value specifies a variable-pitch system font.
DEVICE_DEFAULT_FONT	This value specifies a device-dependent font.
OEM_FIXED_FONT	This value specifies an OEM-dependent fixed font.
SYSTEM_FONT	This value specifies a system font. This font is used to draw menus, dialog box controls, and other text; it is a variable-pitch font width.

SYSTEM_FIXED_FONT	This value specifies a fixed-pitch system font used in Windows versions earlier than 3.0.
DEFAULT_PALETTE	This value specifies a default color palette consisting of the system palette's static colors.

* The gray stock brush objects should be used only in a window that has the CS_HREDRAW and CS_VREDRAW class styles set. Using a gray stock brush in any other style of window can lead to alignment problems.

Note: Since a stock object is owned by the system, the object should not be deleted by your application.

186.3 Returns

GetStockObject() returns a handle to the desired object, if it is successful. Otherwise, it returns NULL.

186.4 Errors

None.

186.5 Cross-References

None.

187 IsGDIObject

187.1 Synopsis

BOOL IsGDIObject(HGDIOBJ hObject);

187.2 Description

The *IsGDIObject()* function determines if a given handle is not a handle to a graphics device interface (GDI) object. The *hObject* parameter is a handle to the object that should be tested. This function should not be used to determine if the given handle is a handle to a GDI object.

187.3 Returns

IsGDIObject() returns TRUE if the given handle may be the handle of a GDI object. *IsGDIObject()* returns FALSE if the handle is not the handle of a GDI object.

187.4 Errors

None.

187.5 Cross-References

None.

188 AnimatePalette

188.1 Synopsis

```
void AnimatePalette(HPALETTE hPalette, UINT First, UINT NumEntries,
        const PALETTEENTRY *EntryArray);
```

188.2 Description

The *AnimatePalette()* function replaces the entries in a logical palette with new entries. The handle of the logical palette is given in the function parameter *hPalette*. The *First* parameter specifies the index number of the first palette entry that is changed in the logical palette. The *NumEntries* parameter specifies the number of palette entries that are changed in the logical palette. The *EntryArray* parameter is a pointer to the first **PALETTEENTRY** structure in an array of **PALETTEENTRY** structures whose values replace the entries in the logical palette.

After calling this function, the application does not have to redraw its client area since the new palette entries will immediately be mapped into the system palette.

AnimatePalette() only replaces the logical palette entries that have the PC_RESERVED flag set.

188.3 Returns

None.

188.4 Errors

None.

188.5 Cross-References

PALETTEENTRY

---

189 CreatePalette

189.1 Synopsis

HPALETTE CreatePalette(LOGPALETTE *LogPalettePtr);

189.2 Description

CreatePalette() creates a logical palette. The *LogPalettePtr* parameter is a pointer to a **LOGPALETTE** structure that contains information about the new logical palette.

When the palette returned by the *CreatePalette()* function is no longer needed, an application should delete the palette with the *DeleteObject()* function.

189.3 Returns

If the *CreatePalette()* function is successful, it returns a handle to the new logical palette. If the *CreatePalette()* function is not successful, it returns NULL.

189.4 Errors

None.

189.5 Cross-References

DeleteObject(), **LOGPALETTE**

---

190 GetNearestColor, GetNearestPaletteIndex

190.1 Synopsis

COLORREF GetNearestColor(HDC hdc, COLORREF ColorRef);

UINT GetNearestPaletteIndex(HPALETTE hPalette, COLORREF ColorRef);

190.2 Description

The *GetNearestColor()* function finds a solid color that can be represented in the given device-context and that is closest to the specified color. The *hdc* parameter specifies the device-context. The *ColorRef* parameter contains the color to match.

GetNearestPaletteIndex() returns the index number of the logical palette entry whose color is closest to the specified color. The *hPalette* parameter is the handle of a logical palette. The *ColorRef* parameter contains the color to match in the logical palette.

190.3 Returns

GetNearestColor() returns a COLORREF value containing the closest solid color match.

GetNearestPaletteIndex() returns the index number of the logical palette entry whose color is closest to the specified color. The index number of the first logical palette entry is zero.

190.4 Errors

None.

190.5 Cross-References

COLORREF, HPALETTE

191 GetPaletteEntries, GetSystemPaletteEntries, SetPaletteEntries

191.1 Synopsis

UINT GetPaletteEntries(HPALETTE hPalette, UINT First, UINT NumEntries,

    PALETTEENTRY *EntryArray);

UINT GetSystemPaletteEntries(HDC hdc, UINT First, UINT NumEntries,

    PALETTEENTRY *EntryArray);

UINT SetPaletteEntries(HPALETTE hPalette, UINT First, UINT NumEntries,

    const PALETTEENTRY *EntryArray);

191.2 Description

The *GetPaletteEntries()* function retrieves a logical palette's palette entry information and stores the information in an array of palette entry structures. The handle of the logical palette is given in the function parameter, *hPalette*. The *First* parameter specifies the index number of the first palette entry to be retrieved from the logical palette. The *NumEntries* parameter specifies the number of palette entries that is retrieved from the logical palette. The *EntryArray* parameter is a pointer to the first **PALETTEENTRY** structure in an array of **PALETTEENTRY** structures and is used to store the logical palette's palette entry information. The array should contain at least as many **PALETTEENTRY** structures as specified in the *NumEntries* parameter.

The *GetSystemPaletteEntries()* function retrieves the system palette's palette entry information and stores the information in an array of palette entry structures. The handle of the device-context is given in the parameter, *hdc*. The *First* parameter specifies the index number of the first palette entry retrieved from the system palette. The *NumEntries* parameter specifies the number of palette entries that are retrieved from the system palette. The *EntryArray* parameter is a pointer to the first **PALETTEENTRY** structure in an array of **PALETTEENTRY** structures and is used to store the system palette's palette entry information. The array should contain at least as many **PALETTEENTRY** structures as specified in the *NumEntries* parameter.

The *SetPaletteEntries()* function replaces the entries in a logical palette with new entries. The handle of the logical palette is given in the function parameter, *hPalette*. The *First* parameter specifies the index number of the first palette entry that is changed in the logical palette. The *NumEntries* parameter specifies the number of palette entries that will be changed in the logical palette. The *EntryArray* parameter is a pointer to the first **PALETTEENTRY** structure in an array of **PALETTEENTRY** structures whose values are replaced by the entries in the logical palette.

If the logical palette specified in the *hPalette* parameter is selected into a device-context when *SetPaletteEntries()* is called, the changes to the logical palette do not take effect until the application calls the *RealizePalette()* function.

191.3 Returns

If the *GetPaletteEntries()* function is successful, it returns the number of entries retrieved from the logical palette. If the *GetPaletteEntries()* function is not successful, it returns zero.

If the *GetSystemPaletteEntries()* function is successful, it returns the number of entries retrieved from the logical palette. If the *GetSystemPaletteEntries()* function is not successful, it returns zero.

If the *SetPaletteEntries()* function is successful, it returns the number of entries set in the logical palette. If the *SetPaletteEntries()* function is not successful, it returns zero.

191.4 Errors

None.

191.5 Cross-References

PALETTEENTRY, *RealizePalette()*

192 GetSystemPaletteUse, SetSystemPaletteUse

192.1 Synopsis

UINT SetSystemPaletteUse(HDC hdc, UINT StaticValue);

UINT GetSystemPaletteUse(HDC hdc);

192.2 Description

The *SetSystemPaletteUse()* function establishes the number of static colors in the system palette. The *hdc* parameter contains a handle to a device-context that supports color palettes. The *StaticValue* parameter contains one of the following values that determines the number of static colors that are contained in the system palette:

SYSPAL_NOSTATIC	This value specifies two static colors - black and white.
SYSPAL_STATIC	This value specifies twenty static colors that will not change when an application realizes a logical palette.

The *GetSystemPaletteUse()* function gets the number of static colors in the system palette. The *hdc* parameter specifies a handle to a device-context that supports color palettes.

192.3 Returns

If the *SetSystemPaletteUse()* function is successful, it returns the previous StaticValue setting.

If the *GetSystemPaletteUse()* function is successful, it returns one of the following values:

SYSPAL_NOSTATIC	This value specifies two static colors - black and white.
SYSPAL_STATIC	This value specifies twenty static colors that will not change when an application realizes a logical palette.

192.4 Errors

None.

192.5 Cross-References

None.

193 RealizePalette

193.1 Synopsis

UINT RealizePalette(HDC hdc);

193.2 Description

The *RealizePalette()* function maps the palette entries of a device-context's palette to the system palette. The *hdc* parameter contains a handle to a device-context.

193.3 Returns

RealizePalette() returns the number of palette entries in the logical palette that are mapped to different entries in the system palette.

193.4 Errors

None.

193.5 Cross-References

None.

194 SelectPalette

194.1 Synopsis

HPALETTE SelectPalette(HDC hdc, HPALETTE hPalette, BOOL bBackground);

194.2 Description

The *SelectPalette()* function assigns a logical palette to a device-context. Any existing logical palette that was previously assigned to the device-context is replaced. The *hdc* parameter contains a handle to a device-context. The *hPalette* parameter contains the handle of the logical palette that will be assigned to the device-context. The *bBackground* parameter determines whether the logical palette is always to be a background palette. The *bBackground* parameter can be one of the following values:

TRUE	The palette is always background palette.
FALSE	If the device-context is attached to a window, the logical palette is a foreground palette when the window has the input focus.

194.3 Returns

If *SelectPalette()* is successful, it returns the handle of the logical palette previously selected into the given device-context. If *SelectPalette()* is not successful, it returns NULL.

194.4 Errors

None.

194.5 Cross-References

None.

195 GetSysColors

195.1 Synopsis

COLORREF GetSysColors(int ElementType);

195.2 Description

The *GetSysColor()* function retrieves the color of the specified display element. A display element is a standard displayable part of a window or control drawn by the system. The *ElementType* parameter specifies the type of display element about which you wish to receive color information. The *ElementType* parameter can be one of the following values:

COLOR_ACTIVEBORDER	This value specifies the active window's border.
COLOR_ACTIVECAPTION	This value specifies the active window's title.
COLOR_APPWORKSPACE	This value specifies the Multiple Document Interface (MDI) application's background color.
COLOR_BACKGROUND	This value specifies the desktop.
COLOR_BTNFACE	This value specifies the push button's face shading.
COLOR_BTNHIGHLIGHT	This value specifies the selected button in a control.
COLOR_BTNSHADOW	This value specifies the push button's edge shading.
COLOR_BTNTEXT	This value specifies the push button's text.
COLOR_CAPTIONTEXT	This value specifies the title bar, size button, and scroll-bar arrow button's text.
COLOR_GRAYTEXT	This value specifies the grayed (dimmed) text; the color will be zero if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	This value specifies the background of selected item in a control.
COLOR_HIGHLIGHTTEXT	This value specifies the text of selected item in a control.
COLOR_INACTIVEBORDER	This value specifies the inactive window border.
COLOR_INACTIVECAPTION	This value specifies the inactive window title.
COLOR_INACTIVECAPTIONTEXT	This value specifies the text color of an inactive title.
COLOR_MENU	This value specifies the menu's background.
COLOR_MENUTEXT	This value specifies the menu's text.

COLOR_SCROLLBAR	This value specifies the scroll-bar's gray area.
COLOR_WINDOW	This value specifies the window's background.
COLOR_WINDOWFRAME	This value specifies the window's frame.
COLOR_WINDOWTEXT	This value specifies the window's text.

195.3 Returns

If *GetSysColor()* is successful, it returns a COLORREF value for the specified type display element.

195.4 Errors

None.

195.5 Cross-References

COLORREF

---

196 SetSysColors

196.1 Synopsis

void SetSysColors(int NumElements, const int *ElementsPtr, const COLORREF *ColorRefPtr);

196.2 Description

The *SetSysColors()* function sets the colors of the specified display elements. A display element is a standard displayable part of a window or control drawn by the system. The *ElementsPtr* parameter is a pointer to the first integer in an array of integers. Each integer in the *ElementsPtr* array specifies a type of display element whose color should be changed. An integer in the *ElementsPtr* array can have one of the following values:

COLOR_ACTIVEBORDER	This value specifies the active window's border.
COLOR_ACTIVECAPTION	This value specifies the active window's title.
COLOR_APPWORKSPACE	This value specifies the Multiple Document Interface (MDI) application's background color.
COLOR_BACKGROUND	This value specifies the desktop.
COLOR_BTNFACE	This value specifies the push button's face shading.
COLOR_BTNHIGHLIGHT	This value specifies the selected button in a control.
COLOR_BTNSHADOW	This value specifies the push button's edge shading.
COLOR_BTNTEXT	This value specifies the push button's text.
COLOR_CAPTIONTEXT	This value specifies the title bar, size button, and scroll-bar arrow button's text.
COLOR_GRAYTEXT	This value specifies the grayed (dimmed) text; the color will be zero if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	This value specifies the background of selected item in a control.
COLOR_HIGHLIGHTTEXT	This value specifies the text of selected item in a control
COLOR_INACTIVEBORDER	This value specifies the inactive window border.
COLOR_INACTIVECAPTION	This value specifies the inactive window title.
COLOR_INACTIVECAPTIONTEXT	This value specifies the text color of an inactive title.
COLOR_MENU	This value specifies the menu's background.
COLOR_MENUTEXT	This value specifies the menu's text.
COLOR_SCROLLBAR	This value specifies the scroll-bar's gray area.

COLOR_WINDOW	This value specifies the window's background.
COLOR_WINDOWFRAME	This value specifies the window's frame.
COLOR_WINDOWTEXT	This value specifies the window's text.

The *ColorRefPtr* parameter is a pointer to the first COLORREF value in an array of COLORREF values. Each COLORREF value in the *ColorRefPtr* array corresponds to a display element type in the index position of the *ElementsPtr* array. The *SetSysColors()* function changes the first display element specified in the *ElementsPtr* array to the color specified by the first COLORREF value in the *ColorRefPtr* array, and so on.

SetSysColors() only changes the system colors for the current session. The system colors are the default colors in the next session.

196.3 Returns

None.

196.4 Errors

None.

196.5 Cross-References

COLORREF

197 UpdateColors

197.1 Synopsis

int UpdateColors(HDC hdc);

197.2 Description

The *UpdateColors()* function updates the client area of the specified device-context matching the current colors in the client area to the system palette. After the system palette has changed, an inactive window with a realized logical palette can call the *UpdateColors()* function to redraw its client area. The *hdc* parameter specifies the device-context. If the client area needs to be updated quickly, this function can be called to redraw the area. However, there will be loss of some color accuracy since this function performs the color translation based on the color of each pixel before the system palette is changed.

197.3 Returns

None.

197.4 Errors

None.

197.5 Cross-References

None.

198 ResizePalette

198.1 Synopsis

BOOL ResizePalette(HPALETTE hndlpal, UINT nEntries);

198.2 Description

The *ResizePalette()* function changes the size of the logical palette identified by the hndlpal parameter.

The *hndlpal* parameter specifies the palette that has to be changed. The *nEntries* parameter specifies the number of palette entries after the palette has been resized. If the *ResizePalette()* function is called to reduce the size of the palette, the remaining entries in the resized palette are not changed. However, if the function is called to increase the size of the palette, the additional palette entries are set to the color black, and the flags for all additional entries are set to zero.

198.3 Returns

If the function is successful it returns a TRUE value. Otherwise, it returns FALSE.

198.4 Errors

None.

198.5 Cross-References

None.

199 CreateMetaFile, CloseMetaFile

199.1 Synopsis

HDC CreateMetaFile(LPCSTR FileName);

HMETAFILE CloseMetaFile(HDC hdc);

199.2 Description

The *CreateMetaFile()* function creates a metafile. Any actions that are performed using the metafile device-context handle returned by the function are stored in the metafile. If the value of the *FileName* parameter is NULL, a memory-based metafile is created to store the actions. If the value of the *FileName* parameter is not NULL, it is assumed to be a pointer to a string containing the name of a disk-based file that is created to store the actions.

After completing the actions to be stored in the metafile, use the *CloseMetaFile()* function to close the metafile. The *hdc* parameter specifies the metafile device-context that should be closed. Once the metafile handle, which is returned by the *CloseMetaFile()* function, is not needed, it should be deleted using the *DeleteMetaFile()* function.

199.3 Returns

If *CreateMetaFile()* is successful, it returns a handle to a metafile device-context. If the *CreateMetaFile()* function is not successful, it returns NULL.

If the *CloseMetaFile()* function is successful, it returns a handle to a metafile. If the *CloseMetaFile()* function is not successful, it returns NULL.

199.4 Errors

None.

199.5 Cross-References

DeleteMetaFile(), PlayMetaFile()

200 CopyMetaFile

200.1 Synopsis

HMETAFILE CopyMetaFile(HMETAFILE hMetaFile, LPCSTR FileName);

200.2 Description

The *CopyMetaFile()* function copies a metafile and returns a handle to the new copy. The *hMetaFile* parameter contains a handle to the metafile to be copied. If the value of the *FileName* parameter is NULL, a memory-based copy of the metafile is created. If the value of the *FileName* parameter is not NULL, it is assumed be a pointer to a string containing the name of a disk-based file that is to hold the copy of the metafile. Once the metafile handle returned by the *CopyMetaFile()* function is not needed, it should be deleted using the *DeleteMetaFile()* function.

200.3 Returns

If the *CopyMetaFile()* function is successful, it returns a handle to a new copy of the metafile. If the *CopyMetaFile()* function is not successful, it returns NULL.

200.4 Errors

None.

200.5 Cross-References

None.

---

201 DeleteMetaFile

201.1 Synopsis

BOOL DeleteMetaFile(HMETAFILE hMetaFile);

201.2 Description

The *DeleteMetaFile()* function invalidates the metafile handle given in the *hMetaFile* parameter. If the metafile handle is associated with a disk-based metafile, the saved file is not deleted by the *DeleteMetaFile()* function.

201.3 Returns

If *DeleteMetaFile()* is successful, it returns TRUE. If *DeleteMetaFile()* is not successful, it returns FALSE.

201.4 Errors

None.

201.5 Cross-References

None.

---

202 EnumMetaFile, EnumMetaFileProc

202.1 Synopsis

BOOL EnumMetaFile(HDC hdc, HMETAFILE hMetaFile, MFENUMPROC EnumMetaFileProc,

LPARAM lParam)

int CALLBACK EnumMetaFileProc(HDC hdc, HANDLETABLE *HandleTablePtr,

METARECORD *MetaRecordPtr, int NumObjects, LPARAM lParam);

202.2 Description

The *EnumMetaFile()* function enumerates the metafile records of the metafile specified in the *hMetaFile* parameter. Each record in the metafile, the handle to a device-context specified in the *hdc* parameter, and the user-defined value of the *lParam* parameter are passed to the user-defined metafile enumeration callback function specified in the *EnumMetaFileProc* parameter. If the callback function returns FALSE, the *EnumMetaFile()* function stops enumerating the metafile records and returns.

The *EnumMetaFileProc()* function is a user-defined, exported, enumeration callback function of type MFENUMPROC whose address is passed to the *EnumMetaFile()* function. *EnumMetaFile()* passes the device-context handle and user-defined value that it originally received when it was called by the *EnumMetaFileProc()* function with the *hdc* and *lParam* parameters. The *MetaRecordPtr* parameter contains a pointer to the next metafile record in the metafile being enumerated. The *HandleTablePtr* parameter contains a pointer to a handle table containing *NumObjects* number of objects.

202.3 Returns

If the *EnumMetaFile()* function enumerates all of the metafile records in the metafile, it returns TRUE. If the *EnumMetaFile()* does not enumerate all of the metafile records in the metafile, it returns FALSE.

EnumMetaFileProc() should return FALSE to abort the enumeration or TRUE to continue the enumeration.

202.4 Errors

None.

202.5 Cross-References

None.

203 GetMetaFile

203.1 Synopsis

HMETAFILE GetMetaFile(LPCSTR FileName);

203.2 Description

The *GetMetaFile()* function returns a metafile handle for a disk-based metafile. The *FileName* parameter should be a pointer to a string containing the name of the file. Once the metafile handle is not needed, it should be deleted using the *DeleteMetaFile()* function.

203.3 Returns

If the *GetMetaFile()* function is successful, it returns a handle to a metafile. If the *GetMetaFile()* function is not successful, it returns NULL.

203.4 Errors

None.

203.5 Cross-References

DeleteMetaFile()

204 GetMetaFileBits, SetMetaFileBits, SetMetaFileBitsBetter

204.1 Synopsis

HGLOBAL GetMetaFileBits(HMETAFILE hMetaFile);

HMETAFILE SetMetaFileBits(HGLOBAL hGlobal);

HMETAFILE SetMetaFileBitsBetter(HGLOBAL hGlobal);

204.2 Description

The *GetMetaFileBits()* function creates and returns a handle to a global block of memory containing all of a metafile's data. The *hMetaFile* parameter should specify a handle to the desired metafile. If *GetMetaFileBits()* is successful, the metafile handle specified by *hMetaFile* is invalid after the function returns and should not be used. When the metafile's data is no longer needed, the global handle's memory should be freed by calling the *GlobalFree()* function.

The *SetMetaFileBits()* function creates a memory-based metafile using a given global block of memory. The *hGlobal* parameter contains a handle to a global block of memory containing all of a metafile's data. If the *SetMetaFileBits()* function is successful, the global handle specified by *hGlobal* is invalid after the function returns and should not be used. Once the metafile handle is not needed, it should be deleted using the *DeleteMetaFile()* function.

The *SetMetaFileBitsBetter()* function creates a memory-based metafile using a given global block of memory. The *hGlobal* parameter contains a handle to a global block of memory containing all of a metafile's data If the *SetMetaFileBits()* function is successful, the global handle specified by *hGlobal* is invalid after the function returns and should not be used. The metafile handle returned by the *SetMetaFileBitsBetter()* function is owned by GDI and not by the application. The application should not delete the metafile handle. This allows applications that use metafiles for object linking and embedding (OLE) to use metafiles that persist after the termination of the application.

204.3 Returns

If the *GetMetaFileBits()* function is successful, it returns a global handle containing the metafile's data. If the *GetMetaFileBits()* function is not successful, it returns NULL.

If the *SetMetaFileBits()* function is successful, it returns a handle to a memory-based metafile. If the *SetMetaFileBits()* function is not successful, it returns NULL.

If the *SetMetaFileBitsBetter()* function is successful, it returns a handle to a memory-based metafile. If the *SetMetaFileBitsBetter()* function is not successful, it returns NULL.

204.4 Errors

None.

204.5 Cross-References

GlobalFree(), DeleteMetaFile()

---

205 PlayMetaFile, PlayMetaFileRecord

205.1 Synopsis

BOOL PlayMetaFile(HDC hdc, HMETAFILE hMetaFile);

void PlayMetaFileRecord(HDC hdc, HANDLETABLE *HandleTablePtr,

METARECORD *MetaRecordPtr, int NumObjects);

205.2 Description

The *PlayMetaFile()* function plays a metafile to a device-context. The *hdc* parameter specifies a handle to the desired device-context. The *hMetaFile* parameter specifies a handle to the desired metafile.

The *PlayMetaFileRecord()* function plays a single metafile's record to a device-context. The *hdc* parameter specifies a handle to the desired device-context. The *MetaRecordPtr* parameter should contain a pointer to the metafile record. The *HandleTablePtr* parameter should contain a pointer to a handle table containing *NumObjects* number of objects.

205.3 Returns

If the *PlayMetaFile()* function is successful, it returns TRUE. If the *GetMetaFileBits()* function is not successful, it returns FALSE.

PlayMetaFileRecord() has no return value.

205.4 Errors

None.

205.5 Cross-References

None.

---

206 GetViewportExt, GetViewportExtEx

206.1 Synopsis

typedef struct tagSIZE {

int cx

int cy

} SIZE;

DWORD GetViewportExt (HDC hdc);

BOOL GetViewportExtEx(HDC hdc, SIZE *Size);

206.2 Description

The *GetViewportExt()* function, using the handle of the device-context found in the *hdc* parameter, retrieves the vertical and horizontal dimensions, in device units, of the device-context's viewport.

The *GetViewportExtEx()* function uses as its parameters the handle of the device-context, *hdc*, and the pointer to the **SIZE** structure. The **cx** and **cy** members of the **SIZE** structure are filled on return. These are expressed in viewport device units.

206.3 Returns

If successful, the *GetViewportExt()* function returns DWORD. The low-order word contains the horizontal dimension of the device-context's viewport, in device units. The high-order word contains the vertical dimension.

If successful, the *GetViewportExtEx()* function returns TRUE. If it is unsuccessful the function returns FALSE.

206.4 Errors

None.

206.5 Cross-References

SetViewportExt(), SetViewportExtEx()

207 GetViewportOrg, GetViewportOrgEx

207.1 Synopsis

DWORD GetViewportOrg(HDC hdc);

BOOL GetViewportOrgEx(HDC hdc, POINT *Point);

207.2 Description

The *GetViewportOrg()* and *GetViewportOrgEx()* functions retrieve the horizontal and vertical coordinates of the origin of the viewport associated with the device-context specified in the *hdc* parameter.

The *GetViewportOrgEx()* function returns the coordinates of the origin of the viewport in the **POINT** structure pointed to by the function’s *Point* parameter.

207.3 Returns

If successful, the *GetViewportOrg()* function returns a DWORD value containing the coordinates of the origin of the viewport. The DWORD’s low-order word contains the horizontal coordinate and the DWORD’s high-order word contains the vertical coordinate.

If successful, the *GetViewportOrgEx()* function returns TRUE, and the **POINT** structure pointed to by the function’s *Point* parameter will contain the coordinates of the origin of the viewport. If it is unsuccessful, the function returns FALSE.

The values returned by both functions are expressed in device units.

207.4 Errors

None.

207.5 Cross-References

SetViewportOrg(), SetViewportExt()

208 OffsetViewportOrg, OffsetViewportOrgEx

208.1 Synopsis

DWORD OffsetViewportOrg(HDC hdc,int XOffset,int YOffset);

BOOL OffsetViewportOrgEx(HDC hdc, int X, int Y, POINT *Point);

208.2 Description

The *OffsetViewportOrg()* and *OffsetViewportOrgEx()* functions offset the origin of the viewport associated with the device-context specified in the *hdc* parameter. The viewport’s new origin is calculated using the following formula:

New origin x-coordinate = Current origin x-coordinate + X parameter

New origin y-coordinate = Current origin y-coordinate + Y parameter

The parameters’ values should be expressed in device units.

The *OffsetViewportOrgEx()* function returns the original coordinates of the origin of the viewport in the **POINT** structure pointed to by the function's *Point* parameter.

208.3 Returns

If successful, the *OffsetViewportOrg()* function returns a DWORD value containing the original coordinates of the origin of the viewport. The DWORD's low-order word contains the horizontal coordinate and the DWORD's high-order word contains the vertical coordinate.

If successful, the *OffsetViewportOrgEx()* function returns TRUE, and the **POINT** structure pointed to by the function's *Point* parameter will contain the original coordinates of the origin of the viewport. If it is unsuccessful, the function returns FALSE.

The values returned by both functions are expressed in device units.

208.4 Errors

None.

208.5 Cross-References

GetViewportOrg(), SetViewportOrg()

---

209 ScaleViewportExt, ScaleViewportExtEx

209.1 Synopsis

DWORD ScaleViewportExt(HDC hdc, int XNum, int Xdenom, int YNum, int YDenom);

BOOL ScaleViewportExtEx(HDC hdc, int XNum, int XDenom, int YNum, int YDenom, SIZE *Size);

209.2 Description

The *ScaleViewPortExt()* and *ScaleViewPortExtEx()* functions scale the extents of the viewport associated with the device-context specified in the *hdc* parameter. The viewport's new extents are calculated using the following formula:

New x-extent = (Current x-extent *XNum parameter) / XDenom parameter

New y-extent = (Current y-extent *YNum parameter) / YDenom parameter

The parameters' values should be expressed in device units.

The *ScaleViewportExtEx()* function returns the viewport's original extents in the **POINT** structure pointed to by the function's *Point* parameter.

209.3 Returns

If successful, the *ScaleViewportExt()* function returns a DWORD value containing viewport's original extents. The DWORD's low-order word contains the x-extent and the DWORD's high-order word contains the y-extent dimension.

If successful, the ScaleViewportExtEx() function returns TRUE, and the **POINT** structure pointed to by the function's *Point* parameter will contain the viewport's original extents. If it is unsuccessful, the function returns FALSE.

The values returned by both functions are expressed in device units.

209.4 Errors

None.

209.5 Cross-References

GetViewportOrg(), SetViewportOrg()

210 SetViewportExt, SetViewportExtEx

210.1 Synopsis

DWORD SetViewportExt(HDC hdc, int Xdim, int Ydim);

BOOL SetViewportExtEx(HDC hdc, int Xdim, int Ydim, SIZE *Size);

210.2 Description

The *SetViewportExt()* and *SetViewportExtEx()* functions set the extents of the viewport associated with the device-context specified in the hdc parameter. The viewport's x-extent is set to the value of the *Xdim* parameter. The viewport's y-extent is set to the value of the *Ydim* parameter. The values of the *Xdim* and *Ydim* parameters are expressed in viewport device units.

The *SetViewportExtEx()* function returns the viewport's original extents in the **POINT** structure pointed to by the function's *Point* parameter.

210.3 Returns

If successful, the SetViewportExt() function returns a DWORD value containing viewport's original extents. The DWORD's low-order word contains the x-extent and the DWORD's high-order word contains the y-extent.

If successful, the *SetViewportExtEx()* function returns TRUE, and the POINT structure pointed to by the function's *Point* parameter will contain the viewport's original extents. If it is unsuccessful, the function returns FALSE.

The values returned by both functions are expressed in device units.

210.4 Errors

None.

210.5 Cross-References

SetViewportOrg(), SetWindowOrgEx()

211 SetViewportOrg, SetViewportOrgEx

211.1 Synopsis

DWORD SetViewportOrg(HDC hdc,int XOrg, int YOrg);

BOOL SetViewportOrgEx(HDC hdc, int XOrg, int YOrg, POINT *Point);

211.2 Description

The *SetViewportOrg()* and *SetViewportOrgEx()* functions set the horizontal and vertical coordinates of the origin of the viewport associated with device-context specified in the *hdc* parameter. The origin's x-coordinate is set to the value of the *XOrg* parameter. The origin's y-coordinate is set to the value of the *YOrg* parameter. The values of the *XOrg* and *YOrg* parameters should be expressed in viewport units.

The *SetViewportOrgEx()* function returns the original horizontal and vertical coordinates of the origin of the viewport in the **POINT** structure pointed to by the function's *Point* parameter.

211.3 Returns

If successful, the *SetViewportOrg()* function returns a DWORD value containing the original horizontal and vertical coordinates of the origin of the viewport. The DWORD's low-order word contains the horizontal coordinate and the DWORD's high-order word contains the vertical coordinate.

If successful, the *SetViewportOrgEx()* function returns TRUE, and the POINT structure pointed to by the function's *Point* parameter will contain the original horizontal and vertical coordinates of the origin of the viewport. If it is unsuccessful, the function returns FALSE.

The values returned by both functions are expressed in device units.

211.4 Errors

None.

211.5 Cross-References

SetWindowOrg(), SetWindowOrgEx()

212 DPtoLP, LPtoDP

212.1 Synopsis

BOOL DPtoLP(HDC hdc, POINT *lppt, int cPoints)

BOOL LPtoDP(HDC hdc, POINT *lppt, int cPoints)

212.2 Description

The *DPtoLP()* function converts device coordinates given in points into logical coordinates for the same device. *LPtoDP()* makes an opposite conversion, it takes logical coordinates and converts them into device coordinates (points). The conversions depend on the current mapping mode for the given device-context, as well as origins and extents for the device window and viewport.

212.3 Returns

Both functions return a TRUE value if they are successful. Otherwise, they return FALSE.

212.4 Errors

None.

212.5 Cross-References

POINT

213 GetWindowExt, GetWindowExtEx

213.1 Synopsis

DWORD GetWindowExt(HDC hdc);

BOOL GetWindowExtEx(HDC hdc, SIZE *lpSize);

213.2 Description

Both *GetWindowExt()* and *GetWindowExtEx()* retrieve horizontal (x) and vertical (y) extents of the window associated with the device-context *hdc*.

GetWindowExt() returns the result as a doubleword value, while *GetWindowExtEx()* function stores the x-extent and y-extent in a **SIZE** structure specified by *lpSize*.

213.3 Returns

GetWindowExt() returns x-extent in the low-order word of the return value and y-extent in the high-order word.

GetWindowExtEx() returns a TRUE value if it is successful. Otherwise, it returns FALSE.

213.4 Errors

None.

213.5 Cross-References

SetWindowExt(), SetWindowExtEx(), **SIZE**

214 GetWindowOrg, GetWindowOrgEx

214.1 Synopsis

DWORD GetWindowOrg(HDC hdc);

BOOL GetWindowOrgEx(HDC hdc, POINT *lpPoint);

214.2 Description

Both the *GetWindowOrg()* and *GetWindowOrgEx()* functions retrieve horizontal (x) and vertical (y) origins of the window associated with the device-context *hdc*. The *GetWindowOrg()* function returns the result as doubleword value, while the *GetWindowOrgEx()* function stores the logical x- and y-coordinates of the window's origin in a **POINT** structure, specified by *lpPoint*.

214.3 Returns

GetWindowOrg() returns the logical x-coordinate of the window's origin in the low-order word of return value and the y-coordinate in the high-order word.

GetWindowOrgEx() returns a TRUE value if it is successful. Otherwise, it returns FALSE.

214.4 Errors

None.

214.5 Cross-References

GetViewportOrg(), GetViewportOrgEx(), **POINT**, *SetWindowOrg(), SetWindowOrgEx()*

---

215 OffsetWindowOrg, OffsetWindowOrgEx

215.1 Synopsis

DWORD OffsetWindowOrg(HDC hdc, int nXOffset, int nYOffset);

BOOL OffsetWindowOrgEx(HDC hdc, int nXOffset, int nYOffset, POINT *lpPoint);

215.2 Description

Both *OffsetWindowOrg()* and *OffsetWindowOrgEx()* modify the window origin for the device-context *hdc*, using horizontal offset *nXOffset* and vertical offset *nYOffset*. Both offsets must be specified in logical units. *OffsetWindowOrg()* returns logical coordinates of the previous window in a doubleword. *OffsetWindowOrgEx()* places the same data into the **POINT** structure pointed to by the *lpPoint* parameter.

215.3 Returns

OffsetWindowOrg() returns the logical x-coordinate of previous window origin in the low-order word of return value and the y-coordinate in the high-order word.

OffsetWindowOrgtEx() returns a TRUE value if it is successful. Otherwise, it returns FALSE.

215.4 Errors

None.

215.5 Cross-References

GetWindowOrg(), GetWindowOrgEx(), **POINT**, *SetWindowOrg(), SetWindowOrgEx()*

---

216 ScaleWindowExt, ScaleWindowExtEx

216.1 Synopsis

DWORD ScaleWindowExt(HDC hdc, int nXNum, int nXDenom, int nYNum, int nYDenom);

BOOL ScaleWindowExtEx(HDC hdc, int nXNum, int nXDenom, int nYNum, int nYDenom,

    SIZE *lpSize);

216.2 Description

Both *ScaleWindowExt()* and *ScaleWindowExtEx()* modify the window extents for the device-context *hdc*, using the horizontal and vertical numerators *nXNum* and *nYNum*, and the horizontal and vertical denominators *nXDenom* and *nYDenom*. The new horizontal and vertical window offsets are calculated using the following formulas:

$nXNew = (nXOld * nXNum) / nXDenom$

$nYNew = (nYOld * nYNum) / nYDenom$

ScaleWindowExt() returns x- and y-extents of the previous window in a doubleword. *ScaleWindowExtEx()* places the same data into the **SIZE** structure pointed to by lpPoint.

216.3 Returns

ScaleWindowExt() returns the logical x-extent of the previous window origin in the low-order word of the return value and the y-extent in the high-order word.

ScaleWindowExtEx() returns a TRUE value if it is successful. Otherwise, it returns FALSE.

216.4 Errors

None.

216.5 Cross-References

GetWindowExt(), GetWindowExtEx(), SetWindowExt(), SetWindowExtEx(), **SIZE**

---

217 SetWindowExt, SetWindowExtEx

217.1 Synopsis

DWORD SetWindowExt(HDC hdc, int nXExtent, int nYExtent);

BOOL SetWindowExtEx(HDC hdc, int nXExtent, int nYExtent, SIZE *lpSize);

217.2 Description

Both the *SetWindowExt()* and *SetWindowExtEx()* set the horizontal (x) and vertical (y) extents of the window associated with the device-context *hdc*. *SetWindowExt()* returns the window's previous extents as a doubleword value. *SetWindowExtEx()* returns these values in a **SIZE** structure specified by *lpSize*.

Calls to *SetWindowExt()* and *SetWindowExtEx()* are ignored if the following mapping modes are set for the device-context *hdc*:

MM_HIENGLISH

MM_LOENGLISH

MM_HIMETRIC

MM_LOMETRIC

MM_TEXT

MM_TWIPS

217.3 Returns

SetWindowExt() returns the previous window logical x-extent in the low-order word of the return value and the y-extent in the high-order word.

SetWindowExtEx() returns a TRUE value if it is successful. Otherwise, it returns FALSE.

217.4 Errors

None.

217.5 Cross-References

GetWindowExt(), GetWindowExtEx(), SetViewportExt(), SetViewportExtEx(), **SIZE**

---

218 SetWindowOrg, SetWindowOrgEx

218.1 Synopsis

DWORD SetWindowOrg(HDC hdc, int nXOrigin, int nYOrigin);

BOOL SetWindowOrgEx(HDC hdc, int nXOrigin, int nYOrigin, POINT *lpPoint);

218.2 Description

Both the *SetWindowOrg()* and *SetWindowOrgEx()* functions set horizontal (x) and vertical (y) window origins of the device-context *hdc*. *SetWindowOrg()* returns the previous window origins as doubleword values. *SetWindowOrgEx()* returns these values in a **POINT** structure specified by *lpPoint*.

218.3 Returns

SetWindowOrg() returns the logical x-coordinate of the previous window origin in the low-order word of return value and the y-coordinate in the high-order word. *SetWindowOrgEx()* returns a TRUE value if it is successful. Otherwise, it returns FALSE.

218.4 Errors

None.

218.5 Cross-References

GetWindowOrg(), GetWindowOrgEx(), **POINT**, *SetViewportOrg(), SetViewportOrgEx()*

---

219 MapWindowPoints

219.1 Synopsis

void MapWindowPoints(HWND hFromWnd, HWND hToWnd, POINT *PointsPtr, UINT NumPoints);

219.2 Description

The *MapWindowPoints()* function maps a point from one window's coordinate space to another window's coordinate space. The *hFromWnd* parameter is the handle of the window from which to map the points. If the value of the *hFromWnd* parameter is NULL or HWND_DESKTOP, the points to be mapped are assumed to be screen coordinates. The *hToWnd* parameter is the handle of the window to which to map the points. If the value of the *hToWnd* parameter is NULL or HWND_DESKTOP, the points are mapped to screen coordinates. The *PointsPtr* parameter is a pointer to an array of **POINT** structures that are mapped. The *NumPoints* parameter specifies the number of points in the array pointed to by *PointsPtr*.

219.3 Returns

None.

219.4 Errors

None.

219.5 Cross-References

POINT

---

220 WindowFromPoint

220.1 Synopsis

HWND WindowFromPoint(POINT Point);

220.2 Description

The *WindowFromPoint()* function identifies the window that is displayed at a specific screen position. The *Point* parameter contains the screen coordinates of the screen position.

WindowFromPoint() does not consider windows that are hidden, disabled, or transparent. The *ChildWindowFromPoint()* function can be used to find windows with those attributes.

220.3 Returns

If successful, the *WindowFromPoint()* function returns the handle of the window that is displayed at the specific screen position. If no window is displayed at the specified screen location, the *WindowFromPoint()* function returns NULL.

220.4 Errors

None.

220.5 Cross-References

ChildWindowFromPoint (), **POINT**

---

221 ChildWindowFromPoint

221.1 Synopsis

HWND ChildWindowFromPoint(HWND hParentWnd, POINT Point);

221.2 Description

The *ChildWindowFromPoint()* function retrieves the handle of the first child window belonging to the given parent window and that is displayed at a specific location in the parent window's client area. The *hParentWnd* parameter is the handle of the parent window. The *Point* parameter is a **POINT** structure containing the location's coordinates in the parent window's client area. Child windows that are disabled, hidden, or transparent are considered during the search.

221.3 Returns

The *ChildWindowFromPoint()* function returns NULL if the specified location is outside of the parent's window.

If the *ChildWindowFromPoint()* function is successful, it will return the window handle of the first child window that qualifies.

If the *ChildWindowFromPoint()* function cannot find a child window that qualifies, it will return the handle of the parent window.

221.4 Errors

None.

221.5 Cross-References

POINT, **WindowFromPoint()**

---

222 ClientToScreen, ScreenToClient

222.1 Synopsis

void ClientToScreen(HWND hWnd, POINT *PointPtr);

void ScreenToClient (HWND hWnd, POINT *PointPtr);

222.2 Description

The *ClientToScreen()* function maps a window's logical coordinates to the screen's device coordinates. The *hWnd* parameter is the handle to the window. The *PointPtr* parameter is a pointer to a **POINT** structure containing the logical coordinates to map to device coordinates. The device coordinates will be saved to the **POINT** structure pointed to by *PointPtr*.

The *ScreenToClient()* function maps the screen's device coordinates to a window's logical coordinates. The *hWnd* parameter is the handle to the window. The *PointPtr* parameter is a pointer to a **POINT** structure containing the device coordinates to map to logical coordinates. The logical coordinates are saved to the **POINT** structure pointed to by *PointPtr*.

222.3 Returns

None.

222.4 Errors

None.

222.5 Cross-References

POINT

223 CombineRgn

223.1 Synopsis

int WINAPI CombineRgn(HRGN hDestRgn, HRGN hSrcRgn1, HRGN hSrcRgn2, int nCombineMode);

223.2 Description

The *CombineRgn()* function combines regions given in *hSrcRgn1* and *hSrcRgn2* using the mode specified in the *nCombineMode* parameter. The resulting region is placed into *hDestRgn*. For all operations, *hDestRgn* and *hSrcRgn1* must already exist (see *CreateRectRgn()*). If an operation uses *hSrcRgn2*, it must also exist.

The following operations are specified using the *nCombineMode* parameter:

RGN_AND	This value specifies the intersection of *hSrcRgn1* and *hSrcRgn2*.
RGN_COPY	This value copies *hSrcRgn1*.
RGN_DIFF	This value specifies the difference of *hSrcRgn1* and *hSrcRgn*.
RGN_OR	This value specifies the union (combined area) of *hSrcRgn1* and *SrcRgn2*.
RGN_XOR	This value specifies the union of *hSrcRgn1* and *hSrcRgn2*, except for any portions that overlap.

All operations place the resulting region into *hDestRgn*.

223.3 Returns

CombineRgn() returns NULLREGION if *hDestRgn* is empty. It returns SIMPLEREGION if it is a simple rectangle or it returns COMPLEXREGION if the region is not a simple rectangle. *CombineRgn()* returns an ERROR if an invalid HRGN is passed or if an error occurs.

223.4 Errors

None.

223.5 Cross-References

CreateRectRgn(), SetRectRgn()

224 CreateEllipticRgn, CreateEllipticRgnIndirect

224.1 Synopsis

HRGN WINAPI CreateEllipticRgn(int nLeft, int nTop, int nRight, int nBottom);

HRGN WINAPI CreateEllipticRgnIndirect(const RECT *lpr);

224.2 Description

The *CreateEllipticRgn()* and *CreateEllipticRgnIndirect()* functions allocate storage for the region structure, initialize it to make an elliptical region, and return a handle to the new region. The ellipse bounds are defined by the *nLeft, nTop, nRight,* and *nBottom* in the *CreateEllipticRgn()* function. For the *CreateEllipticRgnIndirect()* function, the bounds are defined by the **RECT** structure pointed to by *lpr*. When the application is done with the region, *DeleteObject()* is called to remove it.

224.3 Returns

The functions return the handle of the new region, if successful. If the *CreateEllipticRgn()* or *CreateEllipticRgnIndirect()* functions are unsuccessful, they return NULL.

224.4 Errors

None.

224.5 Cross-References

CreateRectRgn(), CreatePolyRgn(), DeleteObject()

225 CreatePolygonRgn, CreatePolyPolygonRgn

225.1 Synopsis

HRGN WINAPI CreatePolygonRgn(const POINT *lpPt, int cPoints, int mode);

HRGN WINAPI CreatePolyPolygonRgn(const POINT *lpPt, const int *lpnPolyCount, int cIntegers,

int mode);

225.2 Description

The *CreatePolygonRgn()* and *CreatePolyPolygonRgn()* functions allocate storage for the region structure, initialize it to make a polygon region, and return a handle to the new region. In the case of the *CreatePolyPolygonRgn()*, the region consists of a series of closed polygons.

The points in the polygon are contained in an array of **POINT** structures pointed to by *lpPt*. Each point specifies the x and y coordinates of a single polygon vertex. The number of points in the **POINT** array is specified by the *cPoints* parameter. If the points do not close the polygon, the system will automatically do so.

For *CreatePolyPolygonRgn()*, the array of **POINTS** contains the vertices for all of the polygons. Each polygon must be closed as the system does not do so. The *lpnPolyCount* is a pointer to an array of integers where each integer specifies the number of points in its respective polygon. The number of integers (polygons) in the *lpnPolyCount* array is specified by the *cIntegers* parameter.

The mode parameter specifies the polygon fill to be ALTERNATE or WINDING. The ALTERNATE mode fills the area between odd and even numbered sides. The WINDING mode uses direction to determine the areas to be filled. When the application is done with the region, it calls *DeleteObject()* (to remove the region).

225.3 Returns

These functions return the handle of the new region, if successful. If the *CreatePolygonRgn()* or *CreatePolyPolygonRgn()* functions are unsuccessful, they return NULL.

225.4 Errors

None.

225.5 Cross-References

CreateRectRgn(), CreateEllipticRgn(), DeleteObject()

226 CreateRectRgn, CreateRectRgnIndirect

226.1 Synopsis

HRGN WINAPI CreateRectRgn(int nLeft, int nTop, int nRight, int nBottom);

HRGN WINAPI CreateRectRgnIndirect(const RECT *lpr);

226.2 Description

The *CreateRectRgn()* and *CreateRectRgnIndirect()* functions allocate storage for the region structure, initialize it to make an rectangular region, and return a handle to the new region.

The rectangle bounds are defined by the *nLeft*, *nTop*, *nRight*, and *nBottom* in the *CreateRectRgn()* function. For the *CreateRectRgnIndirect()* function, the bounds are defined by the **RECT** structure pointed to by *lpr***.**

When the application is done with the region, it calls *DeleteObject()* (to remove it).

226.3 Returns

These functions return the handle of the new region, if successful. If the *CreateRectRgn()* or *CreateRectRgnIndirect()* functions are unsuccessful, they return NULL.

226.4 Errors

None.

226.5 Cross-References

CreateEllipticRgn(), CreatePolyRgn(), DeleteObject(), **RECT**

227 CreateRoundRectRgn

227.1 Synopsis

HRGN WINAPI CreateRoundRectRgn(int nLeft, int nTop, int nRight, int nBottom, int nWidth,

int nHeight);

227.2 Description

The *CreateRoundRectRgn()* function allocates storage for the region structure, initializes it to make a rounded-corner rectangular region, and returns a handle to the new region. The rectangle bounds are defined by the *nLeft*, *nTop*, *nRight*, and *nBottom* in the *CreateRoundRectRgn()* function. The width and height parameters are used to specify the curvature of the corners. When the application is done with the region, it calls *DeleteObject()* (to remove it).

227.3 Returns

These functions return the handle of the new region, if successful. If the *CreateRoundRectRgn()* function is unsuccessful, NULL is returned.

227.4 Errors

None.

227.5 Cross-References

CreateRectRgn(), DeleteObject()

228 EqualRgn

228.1 Synopsis

BOOL WINAPI EqualRgn(HRGN hRgn1, HRGN hRgn2);

228.2 Description

The *EqualRgn()* function compares two regions to determine if they are equal. HRGN *hRgn1* identifies the first region. HRGN *hRgn2* identifies the second region.

228.3 Returns

The function returns TRUE if the regions are identical. Otherwise, it returns FALSE.

228.4 Errors

None.

228.5 Cross-References

CreateRectRgn()

229 GetRgnBox

229.1 Synopsis

int WINAPI GetRgnBox(HRGN hRgn, RECT *lpr);

229.2 Description

The *GetRgnBox()* function gets the bounding rectangle of the specified region. On return, the **RECT** structure pointed to by *lpr* contains the bounding rectangles coordinates.

229.3 Returns

GetRgnBox() returns NULLREGION if *hRgn* is empty. The function returns SIMPLEREGION if it is a simple rectangle, or it returns COMPLEXREGION if the region is not a simple rectangle. Otherwise, ERROR is returned.

229.4 Errors

This function returns an ERROR if an invalid HRGN is passed or if an error occurs.

229.5 Cross-References

CreateRectRgn()

---

230 OffsetRgn

230.1 Synopsis

int WINAPI OffsetRgn(HRGN hRgn, int nXoffset, int nYoffset);

230.2 Description

The *OffsetRgn()* function moves the region a specified distance. The region is moved *nXoffset* units horizontally (positive moves right, negative moves left) and *nYoffset* units vertically (positive moves down, negative moves up). The region retains its size and shape after the move.

230.3 Returns

OffsetRgn() returns NULLREGION if hRgn is empty. The function returns SIMPLEREGION if it is a simple rectangle or it returns COMPLEXREGION if the region is not a simple rectangle.

230.4 Errors

This function returns an ERROR if an invalid HRGN is passed or if an error occurs.

230.5 Cross-References

CreateRectRgn()

---

231 PtInRegion

231.1 Synopsis

BOOL WINAPI PtInRegion(HRGN hRgn, int nXpos, int nYpos);

231.2 Description

The *PtInRegion()* function determines if a point is located within a region. The horizontal coordinate is *nXpos* and the vertical coordinate is *nYpos*.

231.3 Returns

PtInRegion() returns TRUE if the point specified is enclosed by the outline of the region. Otherwise, it returns FALSE.

231.4 Errors

None.

231.5 Cross-References

CreateRectRgn(), RectInRegion()

---

232 RectInRegion

232.1 Synopsis

BOOL WINAPI RectInRegion(HRGN hRgn, const RECT *lpr);

232.2 Description

The *RectInRegion()* function determines if any part of the rectangle is within the boundaries of the region. The *lpr* parameter is a pointer to a **RECT** structure containing the coordinates of the rectangle.

232.3 Returns

RectInRegion() returns TRUE if any point enclosed by the rectangle intersects with the specified region. *RectInRegion()* returns FALSE if no point enclosed by the rectangle intersects with the specified region.

232.4 Errors

None.

232.5 Cross-References

PtInRgn(), CreateRectRgn()

---

233 SetRectRgn

233.1 Synopsis

void WINAPI SetRectRgn(HRGN hRgn, int nLeft, int nTop, int nRight, int nBottom);

233.2 Description

The *SetRectRgn()* function takes the current region structure and changes it into a rectangular area. The *nLeft* and *nTop* parameters specify the top-left corner of the rectangle. The *nRight* and *nBottom* parameters specify the bottom-right corner of the rectangle. The *hRgn* must be a valid region handle.

233.3 Returns

None.

233.4 Errors

None.

233.5 Cross-References

CreateRectRgn()

---

234 ExcludeClipRect

234.1 Synopsis

int ExcludeClipRect(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);

234.2 Description

The *ExcludeClipRect()* function creates a new clipping region that consists of the existing clipping region minus the rectangle, specified by *nLeftRect*, *nTopRect*, *nRightRect* and *nBottomRect*. The right and bottom edges of the given rectangle are not excluded from the resulting clipping region. The width of the rectangle, specified by the absolute value of *nRightRect* - *nLeftRect* must not exceed 32,767 units. This limit also applies to the height of the rectangle, specified by absolute value of *nBottomRect* - *nTopRect*. The right and bottom edges of the rectangle are not excluded from the resulting clipping region.

234.3 Returns

The return value is SIMPLEREGION when the region has no overlapping borders, COMPLEXREGION when the region has overlapping borders, and NULLREGION when the region is empty.

234.4 Errors

In case of error, the return value is ERROR and no region is created.

234.5 Cross-References

IntersectClipRect(), OffsetClipRect(), SelectClipRect()

---

235 IntersectClipRect

235.1 Synopsis

int IntersectClipRect(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);

235.2 Description

The *IntersectClipRect()* function creates a new clipping region by intersecting the current region and a rectangle, specified by *nLeftRect*, *nTopRect*, *nRightRect* and *nBottomRect*. The right and bottom edges of the given rectangle are excluded from the resulting clipping region. The width of the rectangle, specified by the absolute value of *nRightRect - nLeftRect* must not exceed 32,767 units. This limit also applies to the height of the rectangle, specified by absolute value of *nBottomRect - nTopRect*. The right and bottom edges of the rectangle are not excluded from the resulting clipping region.

235.3 Returns

The return value is SIMPLEREGION when the region has no overlapping borders, COMPLEXREGION when the region has overlapping borders, and NULLREGION when the region is empty.

235.4 Errors

In case of error, the return value is ERROR and the current clipping region remains unaffected.

235.5 Cross-References

ExcludeClipRect(), OffsetClipRect(), SelectClipRect()

---

236 OffsetClipRect

236.1 Synopsis

int OffsetClipRgn(HDC hdc, int nXOffset, int nYOffset);

236.2 Description

The *OffsetClipRect()* function moves the clipping region of the given device-context by the offsets, defined by the *nXOffset* and *nYOffset* parameter.

236.3 Returns

The return value is SIMPLEREGION when the region has no overlapping borders, COMPLEXREGION when the region has overlapping borders, and NULLREGION when the region is empty.

236.4 Errors

In case of error, the return value is ERROR and the current clipping regions remain unaffected.

236.5 Cross-References

IntersectClipRect(), ExcludeClipRect(), SelectClipRect()

---

237 SelectClipRect

237.1 Synopsis

int SelectClipRgn(HDC hdc, HRGN hrgn);

237.2 Description

The *SelectClipRect()* function selects a copy of the region, specified by the *hrgn* parameter, as the current clipping region for the given device-context. Because the *SelectClipRect()* function uses a copy of the given region, it can be simultaneously selected by several device-contexts or it can be deleted. The coordinates of the given region are assumed to be in device units.

237.3 Returns

The return value is SIMPLEREGION when the region has no overlapping borders, COMPLEXREGION when the region has overlapping borders, and NULLREGION when the region is empty.

237.4 Errors

In case of error, the return value is ERROR and the current clipping regions remain unaffected.

237.5 Cross-References

IntersectClipRect(), ExcludeClipRect(), OffsetClipRect()

238 GetClipBox

238.1 Synopsis

int WINAPI GetClipBox(HDC hdc, RECT *lprc);

238.2 Description

The *GetClipBox()* function gets the smallest rectangle that encloses the entire clipping region of the device-context specified by *hdc*. The *lprc* parameter points to the **RECT** structure that will contain the rectangle upon returning.

238.3 Returns

The function returns NULLREGION if clipping region is empty, SIMPLEREGION if it is a simple rectangle, or COMPLEXREGION if the region is not a simple rectangle.

238.4 Errors

None.

238.5 Cross-References

GetBoundsRect(), GetRgnBox(), SelectClipRgn()

239 CreateBrushIndirect

239.1 Synopsis

HBRUSH CreateBrushIndirect(LOGBRUSH *StructPtr);

239.2 Description

The *CreateBrushIndirect()* function creates a brush using the brush attributes specified in a **LOGBRUSH** structure. The *StructPtr* parameter is a pointer to the **LOGBRUSH** structure to use when creating the brush.

When the application no longer needs the brush, it should be deleted using the *DeleteObject()* function.

239.3 Returns

If successful, the *CreateBrushIndirect()* function returns the handle to the brush that is created. If not successful, the *CreateBrushIndirect()* function returns NULL.

239.4 Errors

None.

239.5 Cross-References

DeleteObject(), **LOGBRUSH**

240 CreateDIBPatternBrush

240.1 Synopsis

```
typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER    bmiHeader;
        RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

HBRUSH CreateDIBPatternBrush(HGLOBAL hBitmap, UINT ColorsType);

240.2 Description

The *CreateDIBPatternBrush()* function creates a brush from a device-independent bitmap. The *hBitmap* parameter is a handle to a global memory buffer that contains the device-independent bitmap. The *ColorType* parameter specifies the type of data that is stored in the *bmiColors* element of the device-independent bitmap's **BITMAPINFO** structure. The *ColorType* parameter can be one of the following values:

DIB_PAL_COLORS	The b**miColors** element is an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The **bmiColors** element contains literal RGB values.

If the device-independent bitmap is larger than 8 pixels by 8 pixels, only the bits of the first 8 rows of pixels and first 8 columns of pixels in the upper-left corner of the device-independent bitmap will be used to create the brush.

When the application no longer needs the brush, it should be deleted using the *DeleteObject()* function.

240.3 Returns

If successful, the *CreateDIBPatternBrush()* function returns the handle of the brush that is created. If not successful, the *CreateDIBPatternBrush()* function returns NULL.

240.4 Errors

None.

240.5 Cross-References

BITMAPINFO, *DeleteObject()*

241 CreateHatchBrush

241.1 Synopsis

HBRUSH CreateHatchBrush(int PatternType, COLORREF ColorRef);

241.2 Description

The *CreateHatchBrush()* function creates a brush of the specified pattern and color. The *ColorRef* parameter contains the foreground color of the brush that will be used to draw the hatches. The *PatternType* parameter identifies the brush's hatch pattern type and can be one of the following values:

HS_BDIAGONAL	This value specifies a 45-degree upward hatch from left to right.
HS_CROSS	This value specifies a horizontal and vertical crosshatch.
HS_DIAGCROSS	This value specifies a 45-degree crosshatch.
HS_FDIAGONAL	This value specifies a 45-degree downward hatch from left to right.
HS_HORIZONTAL	This value specifies a horizontal hatch.
HS_VERTICAL	This value specifies a vertical hatch.

When the application no longer needs the brush, it should be deleted using the *DeleteObject()* function.

241.3 Returns

If successful, the *CreateHatchBrush()* function returns the handle to the brush that is created. If not successful, the *CreateHatchBrush()* function returns NULL.

241.4 Errors

None.

241.5 Cross-References

COLORREF, *DeleteObject()*

242 CreateSolidBrush

242.1 Synopsis

HBRUSH CreateSolidBrush(COLORREF ColorRef);

242.2 Description

The *CreateSolidBrush()* function creates a brush of the given color. The *ColorRef* parameter contains the desired color of the brush.

When the application no longer needs the brush, it should be deleted using the *DeleteObject()* function.

242.3 Returns

If successful, the *CreateSolidBrush()* function returns the handle to the brush that is created. If not successful, the *CreateSolidBrush()* function returns NULL.

242.4 Errors

None.

242.5 Cross-References

COLORREF, *DeleteObject()*

243 CreatePen, CreatePenIndirect

243.1 Synopsis

HPEN CreatePen(int Style, int Width, COLORREF ColorRef);

HPEN CreatePenIndirect(LOGPEN LogPenPtr);

243.2 Description

The *CreatePen()* function creates a pen of the given style, width, and color. The *ColorRef* parameter contains the desired color of the pen. The *Width* parameter specifies the width of the new pen in logical units. If the *Width* parameter is zero, the pen is one device unit wide. The *Style* parameter specifies the style of the new pen and can be one of the following values:

PS_SOLID	This value specifies a solid pen; the style cannot have a dithered pen color.
PS_DASH[1]	This value specifies a dashed pen that is one logical pixel wide.
PS_DOT1	This value specifies a dotted pen that is one logical pixel wide.
PS_DASHDOT[1]	This value specifies a pen of alternating dashes and dots that is one logical pixel wide.
PS_DASHDOTDOT[1]	This value specifies a pen of alternating dashes and double dots that is one logical pixel wide.
PS_NULL	This value specifies a null pen.
PS_INSIDEFRAME	This value specifies a a pen that can only draw a line inside the frame of a closed shape. The closed shape must be produced by one of the graphics device interface (GDI) output functions that specify a bounding rectangle (for example, *Rectangle()*). If this pen style is used with a GDI output function that does not specify a bounding rectangle, the drawing area of the pen is not limited to the inside of the shape's frame; for this pen style, if the *ColorRef* parameter refers to a color that is not in the logical palette, the pen will be drawn with a dithered color

[1]Pens of these types cannot be greater than one logical pixel wide.

CreatePenIndirect() creates a pen using the width, color, and style information contained in a **LOGPEN** structure. The *LogPenPtr* parameter is a pointer to a **LOGPEN** structure containing the pen's width, color, and style information. The *CreatePenIndirect()* function has the same parameter value rules as the *CreatePen()* function.

243.3 Returns

If successful, the *CreatePen()* function returns the handle to the pen that is created. If not successful, the *CreatePen()* function returns NULL.

If successful, the *CreatePenIndirect()* function returns the handle to the pen that is created. If not successful, the *CreatePenIndirect()* function returns NULL.

243.4 Errors

None.

243.5 Cross-References

COLORREF, *DeleteObject()*, **LOGPEN**, *Rectangle()*

244 GetBrushOrg, GetBrushOrgEx

244.1 Synopsis

DWORD GetBrushOrg(HDC hdc);

BOOL GetBrushOrgEx(HDC hdc, POINT *PointPtr);

244.2 Description

The *GetBrushOrg()* function returns the origin of the brush that is currently selected in the specified device-context. The *hdc* parameter contains the handle to the device-context.

The *GetBrushOrgEx()* function retrieves the origin of the brush that is currently selected in the specified device-context and stores the coordinates in a **POINT** structure. The *hdc* parameter contains the handle to the device-context. The *PointPtr* parameter contains an address of the **POINT** structure in which the brush's origin position will be stored.

A brush's initial origin is at the coordinates (0,0) in the client area.

244.3 Returns

The coordinates retrieved by the *GetBrushOrg()* and *GetBrushOrg()* functions are in device units and are relative to the desktop window's origin.

The low-order word of the *GetBrushOrg()* function's return value contains the brush's current x-coordinate. The high-order word of the *GetBrushOrg()* function's return value contains the brush's current y-coordinate.

If successful, the *GetBrushOrgEx()* function returns TRUE. If not successful, the *GetBrushOrgEx()* function returns FALSE.

244.4 Errors

None.

244.5 Cross-References

POINT

245 SetBrushOrg

245.1 Synopsis

DWORD SetBrushOrg(HDC hdc, int XOrg, int YOrg);

245.2 Description

The *SetBrushOrg()* function specifies the origin that will be given to the next brush that is selected into the specified device-context. The *hdc* parameter contains the handle to the device-context. The *XOrg* parameter contains a value from 0 to 7 that defines the origin's x-coordinate in device units. The *YOrg* parameter contains a value from 0 to 7 that defines the origin's y-coordinate in device units.

If the *SetBrushOrg()* function is not used, a brush's default initial origin is at the coordinates (0,0) in the client area. Stock system objects should not be used with the *SetBrushOrg()* function.

245.3 Returns

The low-order word of the *SetBrushOrg()* function's return value contains the x-coordinate, in device units, of the previous origin. The high-order word of the *SetBrushOrg()* function's return value contains the y-coordinate, in device units, of the previous origin.

245.4 Errors

None.

245.5 Cross-References

None.

246 CreatePatternBrush

246.1 Synopsis

HBRUSH WINAPI CreatePatternBrush(HBITMAP hbmp);

246.2 Description

The *CreatePatternBrush()* function creates a brush pattern from the bitmap specified by *hbmp*. The resulting brush pattern will use up to 8 pixels by 8 pixels of the source bitmap.

246.3 Returns

The function returns a handle of the newly created brush, if successful. Otherwise, NULL is returned.

246.4 Errors

None.

246.5 Cross-References

DeleteObject(), CreateBitmap(), LoadBitmap()

247 Arc, Chord, Pie

247.1 Synopsis

BOOL Arc(HDC hdc, int Left, int Top, int Right, int Bottom, int XStart, int YStart, int XEnd, int YEnd);

BOOL Chord(HDC hdc, int Left, int Top, int Right, int Bottom, int XStart, int YStart, int XEnd,

int YEnd);

BOOL Pie(HDC hdc, int Left, int Top, int Right, int Bottom, int XStart, int YStart, int XEnd, int YEnd);

247.2 Description

The *Arc(), Chord(),* and *Pie()* functions draw the corresponding graphical object into the specified device-context. Each object is drawn using the currently selected pen.

The *Left*, *Top*, *Right*, and *Bottom* parameters specify the bounding rectangle of an Ellipse. The center of the Ellipse forms the common “Center Point” for two lines, the “Start Line” and “End Line.” The parameters *XStart* and *YStart* specify the second point which defines the “Start Line.” The parameters *XEnd* and *YEnd* specify the second point defining the “End Line.” By infinitely projecting these two lines, their intersection with the Ellipse can be found.

The portion of the Ellipse between the two points of intersection is the arc produced by calling the *Arc()* function.

When the two points of intersection are joined by a straight line and combined with the Arc, the resultant closed figure is the chord.

Alternatively, the arc can be combined with the portion of the “Start Line” and “End Line” that extends from the edge of the Ellipse to the “Center Point” of the Ellipse. This forms the closed pie figure produced when the *Pie()* function is called.

247.3 Returns

These functions return a TRUE value if they are successful. Otherwise, they return FALSE.

247.4 Errors

If no output is produced, check the coordinates of the object for intersection with the visible region of the device-context.

247.5 Cross-References

SelectPen()

248 LineDDA, LineDDAProc

248.1 Synopsis

void LineDDA(int XStart, int YStart, int XEnd, int YEnd, LINEDDAPROC lpProc, LONG lParam);

void CALLBACK LineDDAProc(int XPos, int YPos, LONG lParam);

248.2 Description

This mechanism is used to produce the set of points which compose a straight line. The *LineDDA()* function performs the setup and then repeatedly calls the *LineDDAProc()* function for each point. The line is defined by its two end points. The first is *XStart* and *YStart* and the second point is *XEnd* and *YEnd*.

The *LineDDAProc()* function is an application-defined, exported call-back function of type LINEDDAPROC that is provided by the application to digest the stream of points (*XPos*, *YPos*) as they are produced. The address of this function is passed to *MakeProcInstance()* that returns the value to be placed into the *lpProc* parameter. The *lpProc* parameter is passed to *LineDDA()* along with a generic, long parameter, *lParam*, which is forwarded to the *LineDDAProc()* call as its last parameter.

A typical implementation can involve allocating a memory buffer to contain the points. (The address of this buffer is passed to *LineDDA()*, and therefore, *LineDDAProc()* in the *lParam* variable.) The organization of the buffer has to include the buffer handle, its capacity, an index to the first available entry, and space for the points themselves. The *LineDDAProc()* function then takes each point (*XPos*, *YPos*) and stores it in the entry indicated by the index. The index is incremented and compared to the capacity. If the buffer capacity is reached, it can be reallocated to a new, larger size. When the *LineDDA()* function returns, the buffer has been filled with the points.

248.3 Returns

These functions have no return values. The success of the function is determined by whether the memory buffer has been filled with the series of points as designed by the caller.

248.4 Errors

None.

248.5 Cross-References

MakeProcInstance()

249 LineTo, MoveTo, MoveToEx, Polyline

249.1 Synopsis

typedef struct tagPOINT {

int x;

int y;

} POINT;

DWORD MoveTo(HDC hdc, int X, int Y);

BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint);

BOOL LineTo(HDC hdc, int X, int Y);

BOOL Polyline(HDC hdc, LPPOINT lpPoint, int Count);

249.2 Description

The *MoveTo()* function sets the current position in the specified device-context. The *X* and *Y* parameters contain the new position. The previous position is returned in DWORD with the former *X* coordinate in the low-order word and the former *Y* coordinate in the high-order word.

The *MoveToEx()* function behaves the same as *MoveTo()*, except it includes another parameter. The *lpPoint* parameter either returns the pointer to the previous position or if the value of *lpPoint* is NULL the previous position is not returned.

The *LineTo()* function draws a straight line from the current position to the point indicated by the *X* and *Y* parameters which becomes the new position. The line is drawn into the specified device-context using the currently selected pen.

The *Polyline()* function draws a series of straight lines. The *lpPoint* parameter points to an array of **POINT** structures containing *Count* elements. Each line is drawn from a point in the array to a subsequent point. The *Polyline()* function does not update the current position. The line is drawn into the specified device-context using the currently selected pen.

249.3 Returns

The *MoveTo()* function returns the previous position in a DWORD with the former *X* coordinate in the low-order word and the former *Y* coordinate in the high-order word, if it is successful. Although zero may be returned by the *MoveTo()* function if it fails, zero is also a valid return value upon success.

The *LineTo()*, *Polyline()*, and *MoveToEx()* functions return a non-zero value if they are successful. Otherwise, they return zero. In addition, the *MoveToEx()* function either returns the previous position in the **POINT** structure or NULL.

249.4 Errors

None.

249.5 Cross-References

POINT, *SelectPen()*

---

250 Polygon, PolyPolygon

250.1 Synopsis

BOOL Polygon(HDC hdc, LPPOINT lpPoint, int Count);

BOOL PolyPolygon(HDC hdc, LPPOINT lpPoint, LPINT lpCounts, int PolyCount);

250.2 Description

The *Polygon()* and *PolyPolygon()* functions draw closed figures defined by a series of straight lines. The lines are drawn into the device-context specified by the *hdc* parameter and using the currently selected pen. The closed figures are filled using the currently selected brush. The current position in the device-context is not updated when either drawing or filling the polygon.

The *lpPoint* parameter points to an array of **POINT** structures. Each line is drawn from a point in the array to a subsequent point. The last point in the array is connected to the first point automatically.

For the *Polygon()* function, there are *Count* elements in the *lpPoint* array**.**

The *PolyPolygon()* function draws multiple polygons. There are *PolyCount* polygons described by the *lpPoint* array. The *lpCounts* parameter points to an array of integer counts. For each polygon pointed to by *lpPoint*, there is a corresponding count pointed to by *lpCounts*. When the current point count in the current polygon is exhausted, the polygon is closed and filled. The next point in the *lpPoint* array is the first point of the next polygon.

250.3 Returns

The *Polygon()* and *PolyPolygon()* functions return a TRUE value when they are successful. Otherwise, they return FALSE.

250.4 Errors

None.

250.5 Cross-References

SelectPen(), SelectBrush(), Polyline()

251 Ellipse, Rectangle, RoundRect

251.1 Synopsis

BOOL Ellipse(HDC hdc, int Left, int Top, int Right, int Bottom);

BOOL Rectangle(HDC hdc, int Left, int Top, int Right, int Bottom);

BOOL RoundRect(HDC hdc, int Left, int Top, int Right, int Bottom, int Width, int Height);

251.2 Description

The *Ellipse()*, *Rectangle()*, and *RoundRect()* functions draw simple closed figures into the specified device-context using the currently selected pen. The figures are then filled using the currently selected brush.

The *Rectangle()* and *Ellipse()* functions draw respectively a rectangle or an ellipse as defined by the *Left*, *Top*, *Right*, and *Bottom* parameters. The *RoundRect()* function draws a rectangle with rounded corners. The *Left*, *Top*, *Right*, and *Bottom* parameters specify the rectangle and the *Width* and *Height* parameters specify the ellipse that defines the curvature of the corners.

251.3 Returns

These functions return TRUE, if they are successful. Otherwise, they return FALSE.

251.4 Errors

None.

251.5 Cross-References

SelectPen(), SelectBrush()

252 FrameRgn, FillRgn, InvertRgn, PaintRgn

252.1 Synopsis

BOOL FrameRgn(HDC hdc, HRGN hRegion, HBRUSH hBrush, int Width, int Height);

BOOL FillRgn(HDC hdc, HRGN hRegion, HBRUSH hBrush);

BOOL PaintRgn(HDC hdc, HRGN hRegion);

BOOL InvertRgn(HDC hdc, HRGN hRegion);

252.2 Description

The region painting functions all use the device-context specified in the *hdc* parameter and a previously created region indicated by the *hRegion* parameter. Before calling any of these functions, the region must be selected into the device-context by calling *SelectObject()*.

The *InvertRgn()* function inverts the colors (for example, black pixels become white pixels) within the region specified by the *hRegion* parameter. The *PaintRgn()* function fills the specified region using the currently selected brush. The *FillRgn()* function is similar to the *PaintRgn()* function, but instead, uses the brush specified by the *hBrush* parameter rather than the brush currently selected in the device-context.

The *FrameRgn()* function uses the brush specified by the *hBrush* parameter to draw a frame around the specified region. The width and height of the frame in device units is indicated by the *Width* and *Height* parameters.

252.3 Returns

These functions return TRUE when they are successful. Otherwise, they return FALSE.

252.4 Errors

None.

252.5 Cross-References

Create region functions, *SelectObject(), SelectBrush()*

253 DrawFocusRect

253.1 Synopsis

void DrawFocusRect(HDC hdc, LPRECT lpRect);

253.2 Description

When an object has focus, it is surrounded by a highlighted rectangle. The *DrawFocusRect()* function can be used to draw this rectangle. It uses the rectangle specified by the *lpRect* parameter and the device-context specified by the *hdc* parameter. The rectangle is drawn by using an XOR operation and therefore, a second call to this function removes the previously drawn focus rectangle.

253.3 Returns

None.

253.4 Errors

None.

253.5 Cross-References

SelectFocus()

254 FillRect, FrameRect

254.1 Synopsis

```
typedef struct tagRECT {
        int left;
        int top;
        int right;
        int bottom;
} RECT;
int FillRect(HDC hdc, const RECT *RectPtr, HBRUSH hBrush);
int FrameRect(HDC hdc, const RECT *RectPtr, HBRUSH hBrush);
```

254.2 Description

The *FillRect()* function uses the given brush to fill a rectangular area in the device-context. The *hdc* parameter is the handle to the device-context. The *RectPtr* parameter points to a **RECT** structure containing the rectangular area's logical coordinates. The rectangular area's left and top borders are filled. Its bottom and right borders are not filled. The *hBrush* parameter is the handle of the brush to use when filling the rectangular area.

The *FrameRect()* function uses the given brush to draw a frame around a rectangular area in the device-context. The *hdc* parameter is the handle to the device-context. The *RectPtr* parameter points to a **RECT** structure containing the logical coordinates of the rectangular area to be framed. The *hBrush* parameter is the handle of the brush to use when drawing the frame around the rectangular area. The frame drawn around the rectangular area is one logical pixel wide. The interior of the rectangular area is not filled by the *FrameRect()* function.

FillRect() and *FrameRect()* check the coordinates of the rectangular area given in the *RectPtr* parameter. If the value of the **RECT** structure's **bottom** element is less than the value of its **top** element, or if the value of the **RECT** structure's **right** element is less than the value of its **left** element, the functions return without performing their respective actions.

254.3 Returns

None.

254.4 Errors

None.

254.5 Cross-References

RECT

---

255 FloodFill, ExtFloodFill

255.1 Synopsis

BOOL WINAPI FloodFill(HDC hdc, int nXpos, int nYpos, COLORREF clrref);

BOOL WINAPI ExtFloodFill(HDC hdc, int nXpos, int nYpos, COLORREF clrref, UINT fuFillType);

255.2 Description

FloodFill() and *ExtFloodFill()* fill an area of screen using the currently selected brush. The functions begin filling at the logical *x* and *y* coordinates specified by the *nXpos* and nYpos parameters. The coordinate must be within the clipping region of the device context given in the *hdc* parameter.

In the *FloodFill()* function, the *clrref* parameter specifies the color that will be the boundary for the area filled.

The *fuFillType* parameter determines the fill mode to be used, and must be FLOODFILLBORDER or FLOODFILLSURFACE. If the mode is FLOODFILLBORDER, the function behaves the same as the *FloodFill()* function, where *clrref* is the boundary color for the fill area. In the FLOODFILLSURFACE mode, the area is filled outward from the start point as long as the color *clrref* is encountered.

255.3 Returns

The functions return TRUE if successful. If the point is not within the clip region, or filling is invalid, FALSE is returned.

255.4 Errors

None.

255.5 Cross-References

COLORREF, *GetPixel(), SetPixel()*

---

256 GetPixel, SetPixel

256.1 Synopsis

COLORREF WINAPI GetPixel(HDC hdc, int nXpos, int nYpos);

COLORREF WINAPI SetPixel(HDC hdc, int nXpos, int nYpos, COLORREF clrref);

256.2 Description

The *SetPixel()* and *GetPixel()* functions set or retrieve the RGB color value of a pixel respectively. The pixel is located at the logical x and y coordinates specified by the *nXpos* and *nYpos* parameters, and must be within the clipping region of the device context given in the hdc parameter.

The *SetPixel()* function sets the pixel to the closest approximation of the RGB color specified by *clrref.*

If the device does not support raster operations, the *SetPixel()* function will fail. Use the *GetDevCaps()* function with the RC_BITBLT constant to determine if raster operations are supported.

256.3 Returns

The functions return the actual color of the pixel. In the case of *SetPixel()*, that color may be different than the one specified if an approximation was used. If the function fails or the pixel is not within the clipping region, -1 is returned.

256.4 Errors

None.

256.5 Cross-References

COLORREF, *GetDevCapsl()*

257 CreateBitmap, CreateBitmapIndirect

257.1 Synopsis

```
typedef struct tagBITMAP {
        int         bmType;
        int         bmWidth;
        int         bmHeight;
        int         bmWidthBytes;
        BYTE        bmPlanes;
        BYTE        bmBitsPixel;
        LPVOID      bmBits;
} BITMAP;
```

```
HBITMAP CreateBitmap(int nWidth, int nHeight, UINT uiPlanes, UINT uiBitsPerPixel,
        const void *lpvBits);
HBITMAP CreateBitmapIndirect(BITMAP *lpbm);
```

257.2 Description

The *CreateBitmap()* and *CreateBitmapIndirect()* functions create a bitmap of a specified format and dimension.

The *nWidth* and *nHeight* parameters of *CreateBitmap()*, and the *bmWidth* and *bmHeight* fields of the **BITMAP** structure, which are passed as a parameter to *CreateBitmapIndirect()*, determine the size in pixels of the bitmap being created.

The *uiPlanes* and *uiBitsPerPixel* parameters of *CreateBitmap()* correspond to *bmPlanes* and *bmBitsPixel* fields in the **BITMAP** structure passed to *CreateBitmapIndirect()* and define the number of color planes and color bits per display pixel in the bitmap. The *lpvBits* parameter (or *bmBits* field of the **BITMAP** structure) points to an array of initial bit values for the bitmap in the device-dependent representation.

This parameter can be NULL, in which case, the bitmap is not initialized. The function *DeleteObject()* is used to destroy the bitmap created by these functions.

257.3 Returns

These functions return a handle of the created bitmap, if successful. Otherwise, they return NULL.

257.4 Errors

These functions can fail if there is not enough memory to support the color formatting.

257.5 Cross-References

BITMAP, *CreateCompatibleBitmap(), CreateDIBitmap(), SelectObject(), DeleteObject()*

258 CreateCompatibleBitmap, CreateDiscardableBitmap

258.1 Synopsis

```
HBITMAP CreateCompatibleBitmap(HDC hdc, int nWidth, int nHeight);
HBITMAP CreateDiscardableBitmap(HDC hdc, int nWidth, int nHeight);
```

258.2 Description

The *CreateCompatibleBitmap()* and *CreateDiscardableBitmap()* functions create a bitmap that is compatible with the given device-context (DC) specified in the hdc parameter. If hdc is a memory device-context, the bitmap that is created has the same color format (number of color planes and number of bits per display pixel) as the bitmap currently selected. For non-memory device-contexts, the hdc parameter always has the format of the device.

The *nWidth* and *nHeight* parameters define the size, in pixels, of the bitmap.

CreateDiscardableBitmap() creates a bitmap that can be discarded when it is not selected into any DC. If the bitmap has been discarded, an attempt to select it into a DC fails (*SelectObject()* returns zero).

DeleteObject() is used to destroy the bitmap created by these functions.

258.3 Returns

These functions return the handle of the created bitmap, if successful. Otherwise, they return NULL.

258.4 Errors

The operation fails if the function receives invalid parameters or if there is insufficient memory to complete the operation.

258.5 Cross-References

CreateBitmap(), CreateBitmapIndirect(), CreateDIBitmap(), SelectObject(),DeleteObject()

---

259 CreateDIBitmap

259.1 Synopsis

```
typedef struct tagBITMAPINFOHEADER {
        DWORD     biSize;
        LONG      biWidth;
        LONG      biHeight;
        WORD      biPlanes;
        WORD      biBitCount;
        DWORD     biCompression;
        DWORD     biSizeImage;
        LONG      biXPelsPerMeter;
        LONG      biYPelsPerMeter;
        DWORD     biClrUsed;
        DWORD     biClrImportant;
  } BITMAPINFOHEADER;
  typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER     bmiHeader;
        RGBQUAD              bmiColors[1];
} BITMAPINFO;
HBITMAP CreateDIBitmap(HDC hdc, BITMAPINFOHEADER *lpbmih, DWORD dwInit,
        LPVOID lpvBits, BITMAPINFO *lpbmi, UINT uiColorUse);
```

259.2 Description

The *CreateDIBitmap()* function creates a device-specific bitmap from a device-independent specification.

The information about the requested size of the bitmap is taken from the *biWidth* and *biHeight* fields of the *lpbmih* parameter.

The **biPlanes** and **biBitCount** fields of the **BITMAPINFOHEADER** structure, pointed to by *lpbmih*, provide information about the color format. If both **biPlanes** and **biBitCount** are set to 1, a monochrome bitmap is created. Otherwise, the bitmap created has the color format of the device-context specified in the *hdc* parameter. The *dwInit* parameter determines whether the resulting bitmap is initialized. If it is set to CBM_INIT, the *lpvBits* parameter points to an array of bit values. The format of this data is defined by the color format specified in **biBitCount** field of the **BITMAPINFOHEADER** structure, pointed to by *lpbmih*. In this case, the information about the color values for setting the bitmap bits from *lpvBits* data is contained in the *bmiColors* field of **BITMAPINFO**'s *lpbmi* parameter. The *uiColorUse* parameter determines the way the colors are specified in the **bmiColors** array. It can be one of the following:

DIB_PAL_COLORS	The array consists of indices into the logical palette currently realized in the device-context specified by hdc.
DIB_RGB_COLORS	The array consists of RGBQUAD values.

The function *DeleteObject()* should be used to destroy the bitmap created by these functions.

259.3 Returns

This function returns the handle of the created bitmap, if successful. Otherwise, it returns a zero.

259.4 Errors

None.

259.5 Cross-References

CreateBitmap(), CreateBitmapIndirect(), CreateCompatibleBitmap(), CreateDiscardableBitmap(), SelectObject(), DeleteObject()

260 GetBitmapBits, SetBitmapBits

261.1 Synopsis

LONG GetBitmapBits(HBITMAP hBitmap, LONG cbBuffer, LPVOID lpvBits);

LONG SetBitmapBits(HBITMAP hBitmap, DWORD dwSize, LPVOID lpvBits);

261.2 Description

The *GetBitmapBits()* function retrieves and the *SetBitmapBits()* function sets the bits from the bitmap in device-dependent format.

The *hBitmap* parameter specifies the handle of the bitmap used to access the data. In the *GetBitmapBits()* function, the *lpvBits* parameter points to the buffer that receives a copy of bitmap bits. The *cbBuffer* parameter specifies the number of bytes to be copied into this buffer. In *SetBitmapBits(),* the *lpvBits* parameter points to an array that contains the device-dependent bits to be used in the bitmap. The *cbBuffer* parameter specifies the number of bytes contained in this array. The format of the data is determined by the display device. Therefore, the bits should not be operated on directly by the application.

261.3 Returns

These functions return the number of bytes copied to or from the buffer, if successful. Otherwise, they return zero.

261.4 Errors

None.

261.5 Cross-References

GetDIBits(), SetDIBits()

262 GetDIBits, SetDIBits

262.1 Synopsis

```
typedef struct tagBITMAPINFOHEADER {
        DWORD      biSize;
        LONG       biWidth;
        LONG       biHeight;
        WORD       biPlanes;
        WORD       biBitCount;
        DWORD       biCompression;
        DWORD       biSizeImage;
        LONG        biXPelsPerMeter;
        LONG        biYPelsPerMeter;
        DWORD       biClrUsed;
        DWORD       biClrImportant;
} BITMAPINFOHEADER;
typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER        bmiHeader;
        RGBQUAD                 bmiColors[1];
} BITMAPINFO;
typedef BITMAPINFO *LPBITMAPINFO;
int GetDIBits(HDC hdc, HBITMAP hBitmap, UINT uiStartScan, UINT uiScanLines, LPVOID lpvBits,
        LPBITMAPINFO lpbmi, UINT uiColorUse);
int SetDIBits(HDC hdc, HBITMAP hBitmap, UINT uiStartScan, UINT uiScanLines, LPVOID lpvBits,
        LPBITMAPINFO lpbmi, UINT uiColorUse);
```

262.2 Description

The *GetDIBits()* function retrieves the bits from a bitmap in thhe device-independent format. *SetDIBits()* sets the bits into a bitmap in the device-independent format. The *hdc* parameter specifies which device-context to use. The given bitmap does not have to be selected into the device-context. The *hBitmap* parameter defines the bitmap. The *uiStartScan* parameter specifies the first scan line to be copied to or from the buffer. The *uiScanLines* parameter indicates the number of scan lines affected by the operation. The device-independent bitmap data starts in the bottom-left corner, so the scan line numbers are zero-based, starting from the bottom.

In the *SetDIBits()* function, the *lpvBits* parameter points to the buffer containing the device-independent bits to be set into the bitmap. In the *GetDIBits()* function, the *lpvBits* parameter receives a copy of the device-independent bits from the bitmap.

The **lpbmi** structure contains information about the device-independent bitmap, specifically the color format in **biBitCount** field of the **BITMAPINFOHEADER** structure. This value determines the format of the data in the *lpvBits* buffer.

The *uiColorUse* parameter determines the format of the color information in the *bmiColors* array of the *lpbmi* structure. It can be one of the following values:

DIB_PAL_COLORS	The array contains indices into the logical palette currently pointed to by the hdc device-context.
DIB_RGB_COLORS	The array contains the **RGBQUAD** structures.

If the application only needs to retrieve information about the bitmap, for instance, size or color values in a given format, it passes NULL in the *lpvBits* parameter.

262.3 Returns

These functions return the number of scan-lines actually copied, if they are successful. Otherwise, they return zero.

262.4 Errors

None.

262.5 Cross-References

BITMAPINFOHEADER, *SetBitmapBits(), GetBitmapBits(), SetDIBitsToDevice(), StretchDIBits()*

263 SetDIBitsToDevice

263.1 Synopsis

```
typedef struct tagBITMAPINFOHEADER {
        DWORD       biSize;
        LONG        biWidth;
        LONG        biHeight;
        WORD        biPlanes;
        WORD        biBitCount;
        DWORD       biCompression;
        DWORD       biSizeImage;
        LONG        biXPelsPerMeter;
        LONG        biYPelsPerMeter;
        DWORD       biClrUsed;
        DWORD       biClrImportant;
} BITMAPINFOHEADER;
typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER        bmiHeader;
        RGBQUAD                 bmiColors[1];
} BITMAPINFO;
typedef BITMAPINFO *LPBITMAPINFO;
int SetDIBitsToDevice(HDC hdc, int nXDest, int nYDest, int nWidth, int nHeight, int nXSrc, int nYSrc,
        UINT uiStartScan, UINT uiScanLines, LPVOID lpvBits, LPBITMAPINFO lpbmi, UINT uiColorUse);
```

263.2 Description

The *SetDIBitsToDevice()* function sets the device-independent bits directly into the device-context specified in the *hdc* parameter. The *nXDest* and *nYDest* parameters define the logical coordinates of the rectangle in the device-context affected by the operation. The *nXSrc* and *nYSrc* parameters define the lower left corner of the rectangle, in pixels, in the source bitmap. The *nWidth* and *nHeight* parameters define the size, in pixels, of the rectangle in the bitmap.

The *uiStartScan* parameter specifies the number of the starting scan line in the device-independent bitmap. The *uiScanLines* parameter indicates the number of scan lines contained in the *lpvBit* array.

The *lpvBits* parameter points to an array that contains the device-independent bits of the bitmap.

The **BITMAPINFOHEADER** structure (in the **bmiBitsPixel** field of the *lpbmi* parameter) contains the information about the color format of the device-independent bits.

The *uiColorUse* parameter determines the format of the color information in the **bmiColors** array of the **BITMAPINFO** structure, pointed to by *lpbmi*. It can be one of the following values:

DIB_PAL_COLORS	The array contains indices into the logical palette currently realized in the hdc device-context.
DIB_RGB_COLORS	The array contains **RGBQUAD** structures.

263.3 Returns

The function returns the number of scan lines set by the operation, if successful. Otherwise, it returns a zero.

263.4 Errors

None.

263.5 Cross-References

None.

264 StretchDIBits

264.1 Synopsis

```
typedef struct tagBITMAPINFOHEADER {
        DWORD       biSize;
        LONG        biWidth;
        LONG        biHeight;
        WORD        biPlanes;
        WORD        biBitCount;
        DWORD       biCompression;
        DWORD       biSizeImage;
        LONG        biXPelsPerMeter;
        LONG        biYPelsPerMeter;
        DWORD       biClrUsed;
        DWORD       biClrImportant;
} BITMAPINFOHEADER;
typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER        bmiHeader;
        RGBQUAD                 bmiColors[1];
} BITMAPINFO;
typedef BITMAPINFO *LPBITMAPINFO;
int StretchDIBits(HDC hdc, int nXDest, int nYDest, int nWidthDest, int nHeightDest, int nXSrc,
        int nYSrc, int nWidthSrc, int nHeightSrc, LPVOID lpvBits, LPBITMAPINFO lpbmi,
        UINT uiColorUse, DWORD dwRop);
```

264.2 Description

The *StretchDIBits()* function copies the bits from the device-independent bitmap into the device-context specified in *hdc* parameter, stretching or compressing the data, if necessary.

The *nXDest* and *nYDest* parameters specify the logical coordinates of the destination rectangle. The *nWidthDest* and *nHeightDest* parameters determine its size in logical units.

The *nXSrc* and *nYSrc, nWidthSrc* and *nHeightSrc* parameters specify the size and position, in pixels, of the lower-left corner of the source rectangle in the device-independent bitmap.

If the size of the destination rectangle, after being converted to device units, does not match the size of the source rectangle, the bitmap is either stretched or compressed. The stretching-mode attribute of the device-context affects the way this operation is carried out.

StretchDIBits() creates a mirror image of the source bitmap when either *nWidthSrc* and *nWidthDest* (mirroring along the x-axis) or *nHeightSrc* and *nHeightDest* (mirroring along the y-axis) parameters have different signs.

The *lpvBits* parameter points to an array that contains the device-independent bits of the bitmap.

The **BITMAPINFOHEADER** structure (in the **bmiBitsPixel** field of the *lpbmi* parameter) contains the information about the color format of the device-independent bits.

The *uiColorUse* parameter determines the format of the color information in the **bmiColors** array of the **BITMAPINFO** structure, pointed to by *lpbmi*. It can be one of the following values:

DIB_PAL_COLORS	The array contains indices into the logical palette currently pointed to by the hdc device-context.
DIB_RGB_COLORS	The array contains **RGBQUAD** structures.

The *dwROP* parameter specifies a ternary raster-operation code. For the description of this parameter, see *BitBlt()*.

264.3 Returns

The function returns the number of scan lines set by the operation, if successful. Otherwise, it returns zero.

264.4 Errors

None.

264.5 Cross-References

SetDIBits(), SetDIBitsToDevice(), StretchBlt()

265 PatBlt

265.1 Synopsis

BOOL PatBlt(HDC hdc, int nLeft, int nTop, int nWidth, int nHeight, DWORD dwROP);

265.2 Description

The *PatBlt()* function fills the rectangle in the specified device-context with the pattern. The resulting pattern combines the brush selected into the device-context and the bit pattern on the destination device in a way determined by the raster-operation code specified in *dwROP* parameter.

The *nLeft* and *nTop* parameters define the position of the rectangle affected by the operation in the logical coordinates. The *nWidth* and *nHeight* parameters define the size of the rectangle. The *dwROP* parameter can be one of the following values:

PATCOPY	This value copies brush pattern.
PATINVERT	This value combines brush pattern with the destination bitmap using XOR operator.
DSTINVERT	This value inverts the destination bitmap.
BLACKNESS	This value paints the destination in black.
WHITENESS	This value paints the destination in white.

265.3 Returns

This function returns TRUE, if it successful. Otherwise, it returns FALSE.

265.4 Errors

None.

265.5 Cross-References

BitBlt(), StretchBlt()

266 BitBlt, StretchBlt

266.1 Synopsis

BOOL BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight, HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwROP);

BOOL StretchBlt(HDC hdcDest, int nXDest, int nYDest, int nWidthDest, int nHeightDest, HDC hdcSrc,

int nXSrc, int nYSrc, int nWidthSrc, int nHeightSrc,DWORD dwROP);

266.2 Description

The *BitBlt()* and *StretchBlt()* functions copy the bitmap from the source device-context (DC) specified in the *hdcSrc* parameter to the *hdcDest* destination device-context. These functions combine the bits from the source device-context with the brush pattern selected for the destination DC, and with the bits from the destination DC as determined by the raster-operation code specified in *dwROP* parameter.

The *nXDest* and *nYDest* parameters define the logical coordinates of the resulting bitmap in the destination DC. The *nXSrc* and *nYSrc* parameters are the logical coordinates of the source bitmap in the source DC.

The *nWidthSrc* and *nHeightSrc* define the size of the rectangle in logical units. The *nWidthDest* and *nHeightDest* parameters of the *StretchBlt()* function define the dimensions of the destination rectangle in logical units. The mapping-mode transformations of the parameters in both DCs are performed. If the resulting rectangle dimensions do not match, stretching or compressing the source bitmap is performed. This operation is affected by the bitmap stretch-mode attribute set in the destination DC.

StretchBlt() creates a mirror image of the source bitmap if either *nWidthSrc* and *nWidthDest* (mirroring along the x-axis) or *nHeightSrc* and *nHeightDest* (mirroring along the y-axis) parameters have different signs.

If the operation involves bitmaps in different color formats, the color format of the source bitmap is converted to match the destination format. When the source bitmap is monochrome and the destination bitmap is color, the monochrome bitmap is converted by setting the bits set to 1 to the background color of the destination DC, and the bits set to 0 to its background color. Conversion from color to monochrome sets the pixels in the source bitmap that match the background color in the source DC to 1 and all others to 0.

The raster operation codes define the logical operation on the pixels of the source (S), pattern (P), and destination (D) bitmaps using a combination of bitwise AND (&), OR (|), XOR (^) and NOT (~) Boolean operators. These codes can be (but not limited to) one of the following (the value on the right shows the resulting pixel):

BLACKNESS	0
NOTSRCERASE	~(S|D)
NOTSRCCOPY	~S
SRCERASE	S&~D
DSTINVERT	~D
PATINVERT	P^D
SRCINVERT	S^D
SRCAND	S&D
MERGEPAINT	~S|D
MERGECOPY	P&S
SRCCOPY	S
SRCPAINT	S|D
PATCOPY	P
PATPAINT	P|~S|D
WHITENESS	1

266.3 Returns

These functions return TRUE, if the operation is successful. Otherwise, they return FALSE.

266.4 Errors

None.

266.5 Cross-References

PatBlt(), SetStretchBltMode(), GetStretchBltMode()

---

267 GetStretchBltMode, SetStretchBltMode

267.1 Synopsis

int WINAPI GetStretchBltMode(HDC hdc);

int WINAPI SetStretchBltMode(HDC hdc, int fnStretchMode);

267.2 Description

The *GetStretchBltMode()* and *SetStretchBltMode()* functions retrieve or set the bitmap stretching mode of the device-context hdc. The stretch mode determines how data is removed from the bitmaps during a *StretchBlt()* operation. The *SetStretchBltMode()* function will set the mode to the one specified by the *fnStretchMode* parameter.

The following are valid stretch modes:

STRETCH_ANDSCANS	This mode AND's the removed lines with the remaining lines, preserving black pixels over colored or white pixels; this is the default stretch mode.
STRETCH_DELETESCANS	This mode removes the lines without preserving any information.
STRETCH_ORSCANS	This mode OR's the removed lines with the remaining lines, preserving white and colored pixels over black pixels.

267.3 Returns

GetStretchBltMode() returns the current stretch mode. *SetStretchBltMode()* returns the previous stretching mode.

267.4 Errors

None.

267.5 Cross-References

StretchBlt(), StretchDIBits()

---

268 GetBitMapDimension, GetBitMapDimensionEx

268.1 Synopsis

DWORD WINAPI GetBitmapDimension(HBITMAP hbmp);

BOOL WINAPI GetBitmapDimensionEx(HBITMAP hbmp, SIZE *lpSize);

268.2 Description

The *GetBitmapDimension()* function returns the dimensions of the bitmap specified by hbmp.

The *GetBitmapDimensionEx()* function retrives the dimensions of the bitmap specified by *hbmp* and stores the dimension information in the **POINT** structure given in the *lpSize* parameter.

The dimensions are in 0.1 millimeter units.

268.3 Returns

GetBitmapDimension() returns the current dimensions, if they have been set by a previous call to the *SetBitMapDimension()* function. The low-order word of the return value contains the width and the high-order word contains the height. If the dimensions have not been set, the function returns zero.

GetBitmapDimensionEx() returns TRUE if successful and FALSE if not successful.

268.4 Errors

None.

268.5 Cross-References

POINT

---

269 SetBitMapDimension, SetBitMapDimensionEx

269.1 Synopsis

DWORD WINAPI SetBitmapDimension(HBITMAP hbmp, int nWidth, int nHeight);

BOOL WINAPI SetBitmapDimensionEx(HBITMAP hbmp, int nWidth, int nHeight, SIZE *lpSize);

269.2 Description

The *SetBitmapDimension()* function set the dimensions of the bitmap specified by *hbmp* and returns the previous dimensions of the bitmap in the function’s return value. The *nWidth* parameter specifies the new width of the bitmap and the *nHeight* parameter specifies its new height.

The *SetBitmapDimensionEx()* function also sets the dimensions of the bitmap specified by hbmp, but rather than return the previous dimensions of the bitmap via the function’s return value, it stores the old dimensions in a given **POINT** structure. The *nWidth* parameter specifies the new width of the bitmap and the *nHeight* parameter specifies its new height.

All dimensions are in 0.1 millimeter units.

269.3 Returns

SetBitmapDimension() returns the current dimensions. The low-order word of the return value contains the width and the high-order word contains the height.

SetBitmapDimensionEx() returns TRUE if successful and FALSE if not successful.

269.4 Errors

None.

269.5 Cross-References

POINT

---

270 AddFontResource

270.1 Synopsis

int AddFontResource(LPSTR lpstrFileName);

270.2 Description

The *AddFontResource()* function adds a font to the font table that is available to all applications. The *lpstrFileName* parameter points to a null-terminated string that contains a valid file name for a font resource. If the function is successful then the application should broadcast a WM_FONTCHANGE message to all other top-level windows.

270.3 Returns

The *AddFontResource()* function returns the number of fonts added.

270.4 Errors

None.

270.5 Cross-References

RemoveFontResource(), SendMessage()

---

271 RemoveFontResource

271.1 Synopsis

int RemoveFontResource(LPSTR lpstrFileName);

271.2 Description

The *RemoveFontResource()* function deletes a font from the font table that is available to all applications. The *lpstrFileName* parameter points to a null-terminated string that contains a valid file name for a font resource. The font is not immediately removed if it is in use. It is however, removed when there are no applications using the font. If the function is successful, the application should broadcast a WM_FONTCHANGE message to all other top-level windows.

271.3 Returns

The *RemoveFontResource()* function returns a non-zero value, if it is successful, and zero otherwise.

271.4 Errors

None.

271.5 Cross-References

AddFontResource(), SendMessage()

272 CreateFont, CreateFontIndirect

272.1 Synopsis

HFONT CreateFont(int iHeight, int iWidth, int iEscapement, int iOrentation, int iWeight, BYTE bIsItalic,

BYTE bIsUnderlined, BYTE bIsStrikeOut, BYTE bCharSet, BYTE bOutputPrecision,
BYTE bClipPrecision, BYTE bQuality, BYTE bPitchAndFamily, LPSTR lpstrFontName);

HFONT CreateFontIndirect(LOGFONT pLogFont);

272.2 Description

The *CreateFont()* function either creates the font according to the parameters it is passed. The *CreateFontIndirect()* function does the same thing except it uses a structure to encompass the parameters which are individually passed to *CreateFont()*.

The *iHeight* parameter specifies the cell height of the font in logical units, if the value is positive. If this value is negative, the height specified is the character height in logical units. If the *iHeight* value is zero, a default height is used. The *iWidth* parameter specifies the average width of characters. If the parameter is zero, a default width that matches the height is used. The *iEscapement* parameter specifies the escapement angle in tenths of degrees counter-clockwise from the x-axis. This angle represents the angle between the x-axis and the line that goes through the origin of the first and last characters on a line. The *iOrientation* parameter is the same as the *iEscapement* parameter, except it relates to the angle between the x-axis and the line that goes through the base line of a character. The *iWeight* parameter specifies the font weight and can be one of the folowing defined values:

FW_DONTCARE:	0
FW_THIN:	100
FW_EXTRALIGHT:	200
FW_ULTRALIGHT:	200
FW_LIGHT:	300
FW_NORMAL:	400
FW_REGULAR:	400
FW_MEDIUM:	500
FW_SEMIBOLD:	600
FW_DEMIBOLD:	600
FW_BOLD:	700
FW_EXTRABOLD:	800

FW_ULTRABOLD:	800
FW_BLACK:	900
FW_HEAVY:	900

The *bIsItalic*, *IsUnderlined*, and *bIsStrikeOut* parameters indicate if the font is to be italic, underlined, or struck-out respectively. Any non-zero value indicates that the attribute is applied, whereas a zero value indicates the attribute is not to be applied. The *bCharSet* parameter can be one of the following values:

ANSI_CHARSET:	0
DEFAULT_CHARSET:	1
SYMBOL_CHARSET:	2
SHIFTJIS_CHARSET:	128
OEM_CHARSET:	255

Using DEFAULT_CHARSET can create unexpected results and so a specific character set should be specified whenever possible. The *bOutputPrecision* parameter can be one of the following:

OUT_CHARACTER_PRECIS

OUT_DEFAULT_PRECIS

OUT_DEVICE_PRECIS

OUT_RASTER_PRECIS

OUT_STRING_PRECIS

OUT_STROKE_PRECIS

OUT_TT_PRECIS

This parameter is used when a given font has two or more output precisions available. The *bClipPrecision* parameter can be one or more of the following:

CLIP_CHARACTER_PRECIS

CLIP_DEFAULT_PRECIS

CLIP_ENCAPSULATE

CLIP_LH_ANGLES

CLIP_MASK

CLIP_STROKE_PRECIS

CLIP_TT_ALWAYS

The *bClipPrecision* values can be combined together as required. The *bQuality* parameter can be one of the following:

DEFAULT_QUALITY

DRAFT_QUALITY

PROOF_QUALITY

The *bPitchAndFamily* parameter can be one of the following:

DEFAULT_PITCH

FIXED_PITCH

VARIABLE_PITCH

FF_DECORATIVE

FF_DONTCARE

FF_MODERN

FF_ROMAN

FF_SCRIPT

FF_SWISS

The final parameter *lpstrFontName* is a null-terminated string that is the name of the font.

272.3 Returns

The *CreateFont()* function returns a handle to a font, or NULL in case of an error.

272.4 Errors

None.

272.5 Cross-References

DeleteObject(), **LOGFONT**

---

273 EnumFonts, EnumFontsProc

273.1 Synopsis

```
int EnumFonts(HDC hdc, LPSTR lpstrFontName, FONTENUMPROC lpfnFontEnumProc,
        LPARAM UserData);

int CALLBACK EnumFontsProc(LOGFONT *pLogFont, TEXTMETRIC *pTextMetric, int iFontType,
        LPARAM UserData);
```

273.2 Description

The *EnumFonts()* function can be used to enumerate the fonts that are installed on the System.

The *hdc* parameter refers to the device-context for the fonts to be enumerated. The function either enumerates all fonts, if the *lpstrFontName* parameter is NULL, or all the fonts of a particular typeface are passed to it. The *lpfnFontEnumProc* parameter relates to the callback function, described below, which receives the enumerated font information. *UserData* can be anything of *sizeof(LPARAM)* and is passed verbatim to the callback function.

The *EnumFontsProc()* function is a application-defined, callback function of type FONTENUMPROC. The call-back function is called by *EnumFonts()* for each font in the system or until the call-back function asks that the enumeration stop. *EnumFonts()* passes **LOGFONT** and **TEXTMETRIC** information about a font to the *EnumFontsProc()* function via the function's *pLogFont* and *pTextMetric* parameters. If the font is a TrueType font, the *pTextMetric* parameter may point to a **NEWTEXTMETRIC** structure. The font type is passed into the function via the *iFontType* parameter. The value *iFontType* can be one of the following:

DEVICE_FONTTYPE

RASTER_FONTTYPE

TRUETYPE_FONTTYPE

273.3 Returns

EnumFonts() returns the last value passed by the callback function, or zero in case of an error.

EnumFontProc() should return a non-zero value if font enumeration is to continue. It should return zero if the enumerating process is not to continue.

273.4 Errors

None.

273.5 Cross-References

LOGFONT, NEWTEXTMETRIC, TEXTMETRIC

274 EnumFontFamilies, EnumFontFamProc

274.1 Synopsis

int EnumFontFamilies(HDC hdc, LPSTR lpstrFontName, FONTENUMPROC lpfnFontEnumProc,

LPARAM UserData);

int CALLBACK EnumFontFamProc(NEWLOGFONT *pNewLogFont, TEXTMETRIC *pTextMetric,

int iFontType, LPARAM UserData);

274.2 Description

The *EnumFontFamilies()* function can be used to enumerate the fonts that are installed on the system.

The *hdc* parameter refers to the device-context for the fonts to be enumerated. The function either enumerates all fonts, if the *lpstrFontName* parameter is NULL, or the particular font passed to it. The *lpfnFontEnumProc* parameter relates to the callback function, described below, which receives the enumerated font information. *UserData* can be anything of *sizeof(LPARAM)* and is passed verbatim to the callback function.

The *EnumFontFamProc()* function is a application-defined, callback function of type FONTENUMPROC. The call-back function is called by *EnumFontFamilies()* for each font in the system or until the call-back function asks that the enumeration stop. *EnumFontFamilies()* passes **NEWLOGFONT** and **TEXTMETRIC** information about a font to the *EnumFontFamProc()* function via the function's *pLogFont* and *pTextMetric* parameters. If the font is a TrueType font, the *pTextMetric* parameter may point to a **NEWTEXTMETRIC** structure. The font type is past into the function via the *iFontType* parameter. The value *iFontType* can be one of the following:

DEVICE_FONTTYPE

RASTER_FONTTYPE

TRUETYPE_FONTTYPE

The user-defined data originally supplied in the *EnumFontFamilies()* function's *UserData* parameter is passed verbatim to the callback function's *UserData* parameter.

274.3 Returns

EnumFontFamilies() returns the last value passed by the callback function, or zero if it is unsuccessful.

EnumFontFamProc() should return a non-zero value if font enumeration is to continue. It should return zero if the enumerating process is not to continue.

274.4 Errors

None.

274.5 Cross-References

LOGFONT, NEWTEXTMETRIC, TEXTMETRIC

275 GetCharWidth, GetABCCharWidths

275.1 Synopsis

BOOL GetCharWidth(HDC hdc, UINT uiFirstChar, UINT uiLastChar, int *piWidths);

BOOL GetABCCharWidths(HDC hdc, UINT uiFirstChar, UINT uiLastChar, LPABC lpABCWidths);

275.2 Description

The *GetCharWidth()* function is used to find the widths of consecutive characters for the font currently selected into the device-context. The *GetABCCharWidths()* function does the same job except that it only works for TrueType fonts and it returns data in logical units in a more detailed **ABC** structure.

The *uiFirstChar* parameter should be a character which is less than or equal to the *uiLastChar* parameter.

The *piWidths* parameter in *GetCharWidth()* points to an array of integers which has at least (*uiLastChar* - *uiFirstChar* + 1) elements. The *lpABCWidths* parameter in *GetABCCharWidths()* points to an array of **ABC**, which has at least (*uiLastChar* - *uiFirstChar* + 1) elements.

275.3 Returns

This function returns TRUE, if it was successful. It returns FALSE if it is unsuccessful.

275.4 Errors

None.

275.5 Cross-References

None.

---

276 GetFontData

276.1 Synopsis

DWORD GetFontData(HDC hdc, DWORD dwTable, DWORD dwOffset, LPVOID lpvBuffer,

DWORD dwDataSize);

276.2 Description

The *GetFontData()* function gets font-metric data for a scalable font. The *dwTable* parameter specifies which font table to use. If this value is zero, the first font table is used. The *dwOffset* parameter specifies the offset in the particular table specified. If this value is zero, the data retrieved is from the beginning of the table. The *lpvBuffer* parameter points to a memory buffer that is at least *dwDataSize* in size. If the *lpvBuffer* is NULL, the function returns the size of buffer required for the specific font data information requested.

276.3 Returns

This function returns the number of bytes of data that were copied to *lpvBuffer* or the number of bytes that could have been copied if *lpvBuffer* was NULL. This function returns -1 if an error has occurred.

276.4 Errors

None.

276.5 Cross-References

None.

---

277 GetKerningPairs

277.1 Synopsis

int GetKerningPairs(HDC hdc, int iPairs, KERNINGPAIR *pKerningPairs);

277.2 Description

The *GetKerningPairs()* function retrieves the kerning pair information for the font currently selected into the device-context specified in the *hdc* parameter. The *iPairs* parameter specifies the number of elements in the array pointed to by *pKerningPairs*. The *pKerningPairs* parameter points to an array of **KERNINGPAIR** structures with at least *iPairs* elements. If *pKerningPairs* is NULL, the number of kerning pairs in the font is returned.

277.3 Returns

This function returns the number of kerning pairs copied into the array pointed to by *pKerningPairs* or the number of kerning pairs for the currently selected font if *pKerningPairs* is NULL. This function will also return zero if there were no kerning pairs for the font or an error occurred.

277.4 Errors

None.

277.5 Cross-References

KERNINGPAIR

278 GetOutlineTextMetrics

278.1 Synopsis

WORD GetOutlineTextMetrics(HDC hdc, UINT uiDataSize,

OUTLINETEXTMETRIC *pOutlineTextMetrics);

278.2 Description

The *GetOutlineTextMetrics()* function gets the **OUTLINETEXTMETRIC** information for a TrueType font that is currently selected into the device-context passed to the function in the *hdc* parameter. The *uiDataSize* parameter specifies the size of the buffer being pointed to by *pOutlineTextMetrics*. The *pOutlineTextMetrics* parameter points to an **OUTLINETEXTMETRIC** structure that receives the text metric information. If the *pOutlineTextMetrics* parameter is NULL, the function returns the number of bytes of information that could have been returned.

278.3 Returns

The return value is non-zero, if the function is successful. It returns zero, if it is unsuccessful.

278.4 Errors

None.

278.5 Cross-References

GetTextMetric(), **TEXTMETRIC**

279 GetRasterizerCaps

279.1 Synopsis

BOOL GetRasterizerCaps(RASTERIZER_STATUS *pRasterizerStatus, int iBufferSize);

279.2 Description

The *GetRasterizerCaps()* returns flags that indicate whether TrueType fonts have been installed. The *pRasterizerStatus* parameter points to a **RASTERIZER_STATUS** structure and the *iBufferSize* parameter indicates the size of the buffer that is pointed to by the *pRasterizerStatus* parameter.

If the *wFlags* member within the **RASTERIZER_STATUS** structure is TT_AVAILABLE, there is at least one TrueType font available. If the *wFlags* member is set to TT_ENABLED, TrueType fonts have been enabled for the system.

279.3 Returns

This function returns TRUE, if the function is successful. Otherwise, it returns FALSE.

279.4 Errors

None.

279.5 Cross-References

GetOutlineTextMetrics()

280 GetAspectRatioFilter, GetAspectRatioFilterEx

280.1 Synopsis

DWORD GetAspectRatioFilter(HDC hdc);

BOOL GetAspectRatioFilterEx(HDC hdc, SIZE *pAspectRatio);

280.2 Description

The *GetAspectRatioFilter()* function retrieves the aspect ratio filter for the current selection in the *hdc* parameter. The aspect ratio is the relationship between the pixel height and width of a font. The filter allows fonts designed for a particular aspect ratio to be selected. The *SetMapperFlags()* function sets the aspect ratio used by the filter.

The *hdc* parameter is the device-context for which the aspect ratio filter is retrieved. The *pAspectRatio* parameter in the *GetAspectRatioFilterEx()* function points to a **SIZE** structure.

280.3 Returns

The *GetAspectRatioFilter()* function returns the x-coordinate in the low-order word and the y-coordinate in the high-order word.

The *GetAspectRatioFilterEx()* function returns a TRUE value, if the function is successful. Otherwise, it returns FALSE.

280.4 Errors

None.

280.5 Cross-References

SetMapperFlags(), **SIZE**

281 SetMapperFlags

281.1 Synopsis

DWORD SetMapperFlags(HDC hndlDC, DWORD fdwFontMatch);

281.2 Description

The *SetMapperFlags()* function is used to change the method by which the font mapper converts a logical font to a physical font. This function can be called when it becomes necessary to have the font selector attempt to select only a physical font which would exactly match the aspect ratio of the device-context specified by the *hndlDC* parameter. The *fdwFontMatch* parameter identifies whether the font mapper will try to match the font's aspect ratio to the device. If the value of this parameter is ASPECT_FILTERING, the font mapper selects only those fonts whose x-aspect and y-aspect exactly match those of the specified device, ignoring the remaining bits.

If the application uses only raster fonts, *SetMapperFlags()* can be called to make sure that a readable and attractive font is chosen for the given device. If the application uses scalable fonts, *SetMapperFlags()* is generally not used. In the event that no physical font matches the specifications in the logical font, the GDI will choose a new aspect ratio and will find a font that matches this new aspect ratio.

281.3 Returns

This function returns the previous value of the font-mapper flag, if it is successful.

281.4 Errors

None.

281.5 Cross-References

None.

282 DrawText

282.1 Synopsis

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

DrawText(HDC hdc, LPCSTR String, int Count, LPRECT Rect, UINT Flags);

282.2 Description

The *DrawText()* function draws *Count* bytes of *String* to the device-context specified in the *hdc* parameter using the device-context's output position, selected font, background and text colors. If the value of *Count* is -1, *DrawText()* assumes that *String* is null-terminated and outputs all of its characters up to the first null character.

The *Rect* parameter points to the **RECT** structure that provides logical coordinates of the rectangle for text formatting.

DrawText() has the ability to format the output of the text string using the function's *Rect* and *Flags* parameters. The following values for the *Flags* parameter determines how *DrawText()* draws *String*:

DT_BOTTOM	Text is aligned along the bottom of *Rect*; DT_SINGLELINE must be used with this flag.
DT_CALCRECT	*DrawText()* calculates the smallest rectangle needed to bound, but not clip, the output text string and returns the result in the *Rect* parameter. Nothing gets drawn to *hdc* when this flag is used. If DT_CALCRECT is not combined with DT_SINGLELINE, *DrawText()* wraps the text using the width of *Rect* and, if necessary, modifies the bottom of *Rect* to bound the last line of text. If DT_CALCRECT is combined with DT_SINGLELINE, *DrawText()*, if necessary, modifies the right field of the **RECT** structure to bound the last character in the text line.
DT_CENTER	Text is centered horizontally in *Rect*.
DT_EXPANDTABS	*DrawText()* expands any tab characters in the string; tabs are expanded to eight characters per tab if the DT_TABSTOP is not used.
DT_EXTERNALLEADING	*DrawText()* includes the font's external leading value when calculating a line's height; by default, the font's external leading is not considered when calculating the height of a line of text.
DT_LEFT	Text is aligned at the left side of *Rect*.
DT_NOCLIP	This flag notifies *DrawText()* that text drawn outside of the *Rect* boundaries should not be clipped; by default, *DrawText()* clips text that falls outside of the *Rect* boundary.
DT_NOPREFIX	This flag instructs *DrawText()* to draw all "&" characters found in the text string; by default, an "&" character in the text string directs *DrawText()* to underline the next character that follows, and a sequence of two "&" characters (&&) directs *DrawText()* to draw a single & character.
DT_RIGHT	Text aligns at the right side of *Rect*.
DT_SINGLELINE	The text string 's text is not wrapped if it is too long to fit in *Rect*; any carriage returns or linefeeds in the text string are ignored.
DT_TABSTOP	This flag instructs *DrawText()* to use the value of the high-order byte in the *Flags* parameter as the number of characters for each tab. The flag should be used in conjunction with the DT_EXPANDTABS flag. This flag cannot be used with the DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX flags.
DT_TOP	Text is aligned along the top of *Rect*; DT_SINGLELINE must be used with this flag.
DT_VCENTER	Text is centered vertically in *Rect*; DT_SINGLELINE must be used with this flag.
DT_WORDBREAK	If the text string extends past the right edge of *Rect*, the line is broken before the word that extends past *Rect* and the remainder of the text string is drawn on the next line. *DrawText()* by default wraps the output

to the next line when it encounters a carriage return–linefeed sequence. This flag should not be used with the DT_SINGLELINE flag.

If the *SetTextAlign()* function's TA_UPDATECP flag has been set in the DC, *DrawText()* starts drawing the text string at the current output position. The TA_UPDATECP flag also prevents *DrawText()* from wrapping the text string even when the DT_WORDBREAK flag is set.

282.3 Returns

If successful, the *DrawText()* function returns the height of the text.

282.4 Errors

None.

282.5 Cross-References

ExtTextOut(), SetBkColor(), SetTextAlign(), SetTextColor(), TabbedTextOut(), TextOut()

---

283 TextOut, ExtTextOut, TabbedTextOut

283.1 Synopsis

BOOL TextOut(HDC hdc, int XStart, int YStart, LPCSTR String, int Count);

BOOL ExtTextOut(HDC hdc, int XStart, int YStart, UINT Flags, LPRECT Rect, LPCSTR String,

    int Count, int *SpacingArray);

LONG TabbedTextOut(HDC hdc, int XStart, int YStart, LPCSTR String, int Count, int NumTabs,

    int *TabsArray, int TabOrigin);

283.2 Description

The *TextOut()* function draws *Count* bytes of a text string pointed to by the *String* parameter in the specified device-context using the *hdc*'s selected font, background and text colors. If the *SetTextAlign()* function's TA_UPDATECP flag is set, the text string is drawn at the current output position, the *XStart* and *YStart* parameters are ignored, and the current output position is updated.

The *ExtTextOut()* function provides the same basic functionality as the *TextOut()* function and offers three additional parameters to aid in output formatting. The value of the *ExtTextOut() Flags* parameter determines how the *ExtTextOut()* function’s *Rect* parameter is used and may be one or more of the following values:

ETO_CLIPPED	*ExtTextOut()* clips text that is drawn outside of *Rect*.
ETO_OPAQUE	The current background color of the *hdc* parameter is used to fill *Rect* before *String* is drawn.

If the value of the *ExtTextOut()* Flags parameter is zero and the *Rect* parameter's value is NULL, *ExtTextOut()* draws the text string to the hdc parameter without consideration of a bounding rectangle.

The *ExtTextOut() SpacingArray* parameter contains logical unit values, representing the distance between the origins of adjacent character cells, for each character in the text string. The first value in *SpacingArray* represents the spacing between the first and second characters in the text string, the second value in *SpacingArray* represents the spacing between the second and third characters in the text string , etc. The value of *SpacingArray* can be NULL if you want to use the default spacing between adajacent characters.

TabbedTextOut() provides the same basic functionality as the *TextOut()* function and also has three additional parameters to support the expansion of tabs in the text string. The *TabsArray* parameter contains the number of values specified in *NumTabs*. Each value represents a tab stop position, in device units, along the x-axis. The tab stop positions in *TabsArray* are sorted in order of increasing value. The first tab stop position in *TabsArray* corresponds to the first tab character contained in *String*. If the value of the *NumTabs* parameter is zero and the value of the *TabsArray* parameter is NULL, each tab character found in the text string is expanded to eight times the average character width of the selected font in *hdc*. If the value of the *NumTabs* parameter is one, the value of the first tab stop in *TabsArray* is used for each tab character found in *String*. If desired, every tab stop position in *TabsArray* can be adjusted by specifying a value in the *TabOrigin* parameter.

283.3 Returns

TextOut() and *ExtTextOut()* return TRUE if the function is successful, and FALSE if they fail.

TabbedTextOut() returns zero if it is not successful. The return value's low-order word is the string's width and its high-order word is the string's height.

283.4 Errors

None.

283.5 Cross-References

DrawText(), SetBkColor(), SetTextAlign(), SetTextColor()

---

284 GetTextExtent, GetTextExtentPoint, GetTabbedTextExtent

284.1 Synopsis

DWORD GetTextExtent(HDC hdc, LPCSTR String, int Count);

BOOL GetTextExtentPoint(HDC hdc, LPCSTR String, int Count, SIZE *Size);

DWORD GetTabbedTextExtent(HDC hdc, LPCSTR String, int Count, int NumTabs, int *TabsArray);

284.2 Description

The *GetTextExtent()* function determines the width and height, in logical units, of the text string pointed to by the *String* parameter of *Count* bytes using the specified device-context's selected font. The clipping region selected in the *hdc* parameter has no impact on the calculations.

The *GetTextExtentPoint()* function provides the same functionality as *GetTextExtent()*, but returns the width and height values in the **SIZE** structure specified in the function's *Size* parameter.

The *GetTabbedTextExtent()* function provides the same functionality as *GetTextExtent()* and also has two additional parameters to support the expansion of tabs in *String*. The *TabsArray* parameter contains the number of values specified in *NumTabs*. Each value represents a tab stop position, in device units, along the x-axis. The tab stop positions in *TabsArray* are sorted in order of increasing value. The first tab stop position in *TabsArray* corresponds to the first tab character contained in *String*. If the value of the *NumTabs* parameter is zero and the value of the *TabsArray* parameter is NULL, each tab character found in *String* is expanded to eight times the average character width of the selected font in *hdc*. If the value of the *NumTabs* parameter is one, the value of the first tab stop in *TabsArray* is used for each tab character found in *String*.

Regardless of which of the three functions that you use, it is important to remember that some devices kern characters. As a result, the sum of the text extent of each character in *String* may not equal the value returned when the entire *String*'s extent is calculated.

284.3 Returns

If successful, the *GetTextExtent()* and *GetTabbedTextExtent()* functions return the *String* width in the low-order word of the return value and the *String* height in the high-order word of the return value.

If successful, the *GetTextExtentPoint()* function returns TRUE. If unsuccessful, the *GetTextExtentPoint()* function returns FALSE.

284.4 Errors

None.

284.5 Cross-References

SIZE

---

285 GetTextAlign, SetTextAlign

285.1 Synopsis

UINT SetTextAlign(HDC hdc, UINT Flags);

UINT GetTextAlign(HDC hdc);

285.2 Description

The *SetTextAlign()* function sets the text alignment characteristics for the specified device-context (DC). *GetTextAlign()* returns the specified device-context's text alignment characteristics.

A device-context's text alignment characteristics are specified as a single value in the *Flags* parameter that is comprised of one or more text alignment flags OR'ed together. The text alignment flags determine the relationship between a point (the current output position or coordinates given to a text output function) and a rectangle that bounds the text to be drawn. The text's bounding rectangle is determined by the adjacent character cells of the text string.

One or more of the following text alignment flags can be OR'ed together to define a DC's text alignment characteristics (only one flag value from each group should be used):

Group 1: x-axis text alignment:

TA_CENTER	The point is aligned with the horizontal center of the bounding rectangle.
TA_LEFT	This is the default text alignment setting for this group; the point is aligned with the left side of the bounding rectangle.
TA_RIGHT	The point is aligned with the right side of the bounding rectangle.

Group 2: y-axis text alignment:

TA_BASELINE	The point is aligned with the base line of the selected font.
TA_BOTTOM	The point is aligned with the bottom of the bounding rectangle.
TA_TOP	This is the default text alignment setting for this group; the point is aligned with the top of the bounding rectangle.

Group 3: Output position updating:

TA_NOUPDATECP	This is the default text alignment setting for this group; the output position is not automatically updated after each call to a text output function.
TA_UPDATECP	The x-axis output position is automatically updated after each call to a text output function. The new position begins at the right side of the text's bounding rectangle. If this flag is set, the x-position/y-position output parameters given in the text output functions are ignored.

285.3 Returns

If the *SetTextAlign()* function is successful, it returns the specified device-context's previous text alignment characteristics before they were changed by the function. The return value's low-order byte contains the horizontal setting and the return value's high-order byte contains the vertical setting. If the *SetTextAlign()* function is not successful, it returns zero.

The *GetTextAlign()* function returns the specified device-context's text alignment characteristics.

285.4 Errors

None.

285.5 Cross-References

None.

286 SetTextColor, GetTextColor

286.1 Synopsis

COLORREF SetTextColor(HDC hdc, COLORREF Color);

COLORREF GetTextColor(HDC hdc);

286.2 Description

Each device-context has an active text color that is used when drawing text in the device-context (DC) and when bitmaps are converted between a monochrome and color device-context. The *SetTextColor()* function sets the specified device-context's (DC) active text color to *Color*. If *Color* cannot be shown in the device-context, the DC's closest available color is used instead. The function *GetTextColor()* can be used to determine a device-context's active text color.

286.3 Returns

If the *SetTextColor()* function is successful, it returns the specified device-context's previous active text color before it was changed by the function.

The *GetTextColor()* function returns the specified device-context's active text color.

286.4 Errors

None.

286.5 Cross-References

RGB macro

---

287 GetTextCharacterExtra, SetTextCharacterExtra

287.1 Synopsis

int SetTextCharacterExtra(HDC hdc, int ExtraSpace);

int GetTextCharacterExtra(HDC hdc);

287.2 Description

Each device-context (DC) has an active extra-horizontal character space value that is used when drawing text into the DC. The extra space is added to the horizontal space used to separate each character in a string drawn to the DC. By default, a DC's extra-horizontal character space value is zero.

The *SetTextCharacterExtra()* function sets the specified device-context's active extra-horizontal character space value *ExtraSpace*. If the DC's mapping mode is a mapping mode other than MM_TEXT, the value of *ExtraSpace* is transformed and rounded to the nearest pixel.

The *GetTextCharacterExtra()* function can be used to determine a device-context's active extra-horizontal character space value.

287.3 Returns

If the *SetTextCharacterExtra()* function is successful, it returns the specified device-context's previous extra-horizontal character space value before it was changed by the function.

The *GetTextCharacterExtra()* function returns the specified device-context's active extra-horizontal character space value.

287.4 Errors

None.

287.5 Cross-References

None.

---

288 GetTextFace

288.1 Synopsis

int GetTextFace(HDC hdc, int Count, LPSTR FaceName);

288.2 Description

Each device-context has an active font that is used when drawing text in a device-context. *GetTextFace()* can be used to determine the face name of a device-context's active font. Given the handle to the device-context, *hdc*, the function copies *Count* bytes of the active font's face name into the *FaceName* buffer.

288.3 Returns

If *GetTextFace()* is successful, it returns the number of bytes that were copied into the *FaceName* buffer, not including the NULL termination character. If the *GetTextFace()* function is not successful, the function returns zero.

288.4 Errors

None.

288.5 Cross-References

None.

---

289 GetTextMetrics

289.1 Synopsis

BOOL GetTextMetrics(HDC hdc, TEXTMETRIC *TextMetric);

289.2 Description

Each device-context has an active font that is used when drawing text into a device-context. *GetTextMetrics()* can be used to retrieve the font metric information about a device-context's active font. Given the handle to the device-context, *hdc*, the function returns information about the device-context's active font in the structure **TEXTMETRIC**.

289.3 Returns

If the *GetTextMetrics()* function is successful, it returns TRUE. Otherwise, the function returns FALSE.

289.4 Errors

None.

289.5 Cross-References

None.

---

290 GrayString, GrayStringProc

290.1 Synopsis

BOOL GrayString(HDC hdc, HBRUSH Brush, GRAYSTRINGPROC DrawProc,

LPARAM DrawProcData, int Count, int x, int y, int cx, int cy);

CALLBACK GrayStringProc HDC hdc, LPARAM DrawProcData, int Count);

290.2 Description

The *GrayString()* function draws gray text in a specified device-context, *hdc*, using the brush handle specified in the *Brush* parameter and the solid gray system color defined by the constant COLOR_GRAYTEXT.

GrayString()'s *DrawProc* parameter is a pointer to a user-defined callback function of type GRAYSTRINGPROC. *GrayString()* calls this callback function to draw the string to a device-context (DC). The *DrawProc()* callback function receives a handle to a device-context created by *GrayString()*, the user-defined data in the *DrawProcData* parameter, and a count of how many characters to print in the *Count* parameter. The callback function should only draw text relative to the 0,0 position in the DC.

If the value of the *GrayString()* function's *DrawProc* parameter is NULL, the value of the *GrayString()* function's *DrawProcData* parameter is a pointer to the string to be drawn. The *GrayString()* function uses the *TextOut()* function to draw the string in the DC. If the value of *DrawProc* is not NULL, the *DrawProcData* parameter is assumed to be the data that the *DrawProc()* callback function needs in order to draw the string in the DC.

The *GrayString()* function only draws *Count* bytes of the string to the device-context. If the value of the *Count* parameter is zero, the *GrayString()* function's *DrawProcData* parameter is assumed to be a pointer to the string to be drawn and the *GrayString()* function attempts to calculate the string's length. If the value of the *Count* parameter is negative and the *GrayString()* function's *DrawProc* callback function returns a value of zero, the string drawn to the specified DC, but not in a gray color.

The *GrayString()* parameters *x*, *y*, *cx*, and *cy* define the logical coordinates of a rectangle that bound the drawn string. The *x* and *y* parameters form the initial output position used for drawing the string. The *cx* parameter is the width of the bounding rectangle. The *cy* parameter is the height of the bounding rectangle from the initial output position. If the value of either the *cx* or *cy* parameter is zero, the *DrawProcData* parameter is assumed to be a pointer to the string to be drawn and the *GrayString()* function attempts to calculate the *cx* or *cy* value for the string.

The specified device-context's active mapping mode must be set to MM_TEXT when using the *GrayString()* function.

290.3 Returns

If the *GrayString()* function is successful, it returns TRUE. If an out-of-memory error occurs, or if the *DrawProc* callback function, or the *TextOut()* function returns zero, the *GrayString()* function returns FALSE.

If the *GrayStringProc()* callback function is successful, it returns a value of TRUE. Otherwise, it returns FALSE.

290.4 Errors

None.

290.5 Cross-References

SetTextColor(), TextOut()

291 SetTextJustification

291.1 Synopsis

int SetTextJustification(HDC hdc, int NumExtraSpace, int NumBreakChars);

291.2 Description

The *SetTextJustification()* function adds a space to the break characters in a string. The *hdc* parameter identifies the device-context. The *NumExtraSpace* parameter identifies the extra space in logical units that has to be added to the line of the text. If the mapping mode in the device-context is not MM_TEXT, the value of this parameter is converted to the current mapping mode and is rounded to nearest device unit. The *NumBreakChars* parameter identifies the number of break characters in the line. The break character is usually a space character (ASCII 32) but it may be specified as any other character by the font.

The font's break character can be retrieved using the function *GetTextMetrics()*. A function call to write text-output after a call to *SetTextJustification()* will distribute the extra space evenly among the specified number of break characters. The application can determine the value of the *NumExtraSpace* parameter by subtracting the return value of a call to *GetTextExtent()* from the desired string width after alignment. The function *SetTextJustification()* can also be used to align a line that contains multiple runs in different fonts. This is done by creating each line seperately and writing each run seperately. Due to rounding errors that can occur during alignment, the system maintains a running error term which defines the current error. When doing multiple runs the function *GetTextExtent()* will use this term to compute the extent of the next run. This error term must be reset at the end of alignment of each line. This is done by calling the function *SetTextJustification()* with the *NumExtraSpace* parameter set to zero.

291.3 Returns

The function is successful it returns 1. Otherwise, it returns zero.

291.4 Errors

None.

291.5 Cross-References

None.

292 AbortDoc

292.1 Synopsis

int AbortDoc(HDC hdc);

292.2 Description

The *AbortDoc()* function terminates the current print job. The *hdc* parameter is the handle to the printer device-context used for the current print job.

292.3 Returns

If successful, the *AbortDoc()* function returns a value equal to or greater than zero. If unsuccessful, the *AbortDoc()* function returns a value less than zero.

292.4 Errors

None.

292.5 Cross-References

EndDoc(), SetAbortProc(), StartDoc()

293 DeviceCapabilities

293.1 Synopsis

```
typedef struct tagDEVMODE {
        char        dmDeviceName[CCHDEVICENAME];
        UINT        dmSpecVersion;
        UINT        dmDriverVersion;
        UINT        dmSize;
        UINT        dmDriverExtra;
        DWORD       dmFields;
              int   dmOrientation;
              int   dmPaperSize;
              int   dmPaperLength;
              int   dmPaperWidth;
              int   dmScale;
              int   dmCopies;
              int   dmDefaultSource;
              int   dmPrintQuality;
              int   dmColor;
              int   dmDuplex;
              int   dmYResolution;
              int   dmTTOption;
} DEVMODE;
```

```
DWORD DeviceCapabilities(LPSTR Device, LPSTR Port, WORD Capability, LPSTR Output,
        LPDEVMODE dm);
```

293.2 Description

The *DeviceCapabilities()* function calls the printer device driver to determine its properties. The *Device* parameter is the name of the printer device. The *Port* parameter is the name of the port. *Device* and *Port* are NULL terminated strings.

The *Capability* parameter retrieves the capabilities of the device. This parameter has one of the following values:

DC_BINNAMES	This value performs a copy of the array containing the paper bins. The array has the form of char *PaperBins[BinMax][BinName]* and *BinName* is 24; if the *Output* parameter is NULL, the return value is the number of required bin entries, or number of bins copied.
DC_BINS	This value copies a list of the available bins to the *Output* parameter in the unsigned short array. If the *Output* parameter is NULL, the return value is the number of supported bins.
DC_COPIES	This value returns the maximum number of copies this device can print.
DC_DRIVER	This value retrieves the printer driver version number.
DC_DUPLEX	This value retrieves the printer driver duplex support level; if *Output* is 1, the printer can print in duplex mode, otherwise *Output* is zero.
DC_ENUMRESOLUTIONS	This value retrieves list of all available resolutions. If *Output* is NULL, it lists all available resolution configurations. Resolutions are pairs of LONG integers representing the horizontal and vertical resolutions (in dots per inch).
DC_EXTRA	This value retrieves the number of bytes for the device-specific portion of the **DEVMODE** structure.
DC_FIELDS	This value retrieves the *dmFields* member of the **DEVMODE** structure; the *dmFields* member indicates which fields in the device-independent portion of the structure are supported by the printer driver.
DC_FILEDEPENDENCIES	This value retrieves the list of files that need to be loaded when the printer device driver is installed. If the *Output* parameter is NULL, the function returns the number of files; otherwise, *Output* points to an array of filenames in the form *char FileName[64]*, where *FileName* is null-terminated.
DC_MAXEXTENT	This value retrieves the **POINT** structure with the maximum paper size that the *dmPaperLength* and *dmPaperWidth* members of the **DEVMODE** structure can specify .
DC_MINEXTENT	This value retrieves **POINT** structure with the minimum paper size that the *dmPaperLength* and *dmPaperWidth* members of the **DEVMODE** structure can specify.
DC_ORIENTATION	This value retrieves the relationship between portrait and landscape orientations, for example, the number of degrees that the portrait orientation is rotated counterclockwise to make landscape orientation. The return value can be one of the following:

0	no landscape orientation
90	portrait is rotated 90 degrees to produce landscape
270	portrait is rotated 270 degrees to produce landscape

DC_PAPERNAMES	This value retrieves a list of supported paper names. If *Output* is NULL, the function returns the number of paper sizes available; otherwise, *Output* points to an array for the paper names in the form, *char PaperNames[64]*. Note that each paper name is a null-terminated.

DC_PAPERS	This value retrieves a list of supported sizes of sheets of paper. The function copies this list to *Output* as a unsigned short array and returns the number of entries in the array. If *Output* is NULL, the function returns the number of supported paper sizes to allow the application to allocate a buffer with the correct size. For more information on paper sizes, see the description of the *dmPaperSize* member of the **DEVMODE** structure.
DC_PAPERSIZE	This value retrieves the array of dimensions of all supported paper sizes, in tenths of a millimeter, to an array in the **POINT** structures pointed to by *Output*; the width and length of a paper size are returned as if the paper was in the DMORIENT_PORTRAIT orientation.
DC_SIZE	This value retrieves the *dmSize* member of the printer driver's **DEVMODE** structure.
DC_TRUETYPE	This value specifies whether this device can use TrueType fonts. The return value is: DCTT_BITMAP This device can print TrueType fonts as graphics. DCTT_DOWNLOAD This device can download TrueType fonts. DCTT_SUBDEV This device can substitute device fonts with TrueType fonts.
DC_VERSION	Retrieves printer driver specification version.

The *Output* parameter is a pointer to an array of bytes, which formats the output depending on the *Capability* parameter. If *Output* is zero, *DeviceCapabilities()* returns the number of bytes required for the output data.

This *dm* parameter is a pointer to a **DEVMODE** structure. If this parameter is NULL, the *DeviceCapabilities()* function retrieves the current default initialization values for the specified printer driver; otherwise, the function retrieves the values contained in the structure to which *dm* points.

293.3 Returns

If successful, the *DeviceCapabilities()* function returns a value which depends on the *Capability* parameter. If not successful, the *DeviceCapabilities()* function returns -1.

293.4 Errors

None.

293.5 Cross-References

DEVMODE, *GetProcAddress(), LoadLibrary()*

294 DeviceMode

294.1 Synopsis

void DeviceMode(HWND Wnd, HANDLE Module, LPSTR Device, LPSTR Output);

294.2 Description

The *DeviceMode()* function sets the printing mode for device. The *Wnd* parameter is the window that was used to create the dialog box for selecting the parameters to the *DeviceMode()* function. The *Module* parameter is the handle to the printer-driver module. This handle should be retrieved by using the *GetModuleHandle()* or the *LoadLibrary()* function. The *Device* parameter is a pointer to the name of the device. *Device* is a null-terminated string. *Output* is the pointer to the name of the physical output media, like file or port.

294.3 Returns

None.

294.4 Errors

None.

294.5 Cross-References

GetDC(), ExtDeviceMode(), GetModuleHandle(), LoadLibrary()

---

295 EndDoc

295.1 Synopsis

int EndDoc(HDC hdc);

295.2 Description

The *EndDoc()* function closes the current print job, if that print job was successful. The *hdc* parameter is the handle to the printer device-context used for the current print job.

Note: Do not use this function with metafiles.

295.3 Returns

If successful, *EndDoc()* returns a value greater than or equal to zero. If it is unsuccessful, the EndDoc() function returns a value less than zero.

295.4 Errors

None.

295.5 Cross-References

AbortDoc(), Escape(), StartDoc()

---

296 EndPage

296.1 Synopsis

int EndPage(HDC hdc);

296.2 Description

The *EndPage()* function informs the output device about the end of the current page. The *hdc* parameter is the handle to the device-context used for the current print job.

296.3 Returns

If successful, the *EndPage()* function returns value greater than or equal to zero. If unsuccessful, the *EndPage()* function returns a value less than zero.

296.4 Errors

If the *EndPage()* function is not successful, the return value is one of the following:

SP_ERROR	Any error not described below.
SP_APPABORT	The current job was terminated.
SP_USERABORT	The user canceled the print job.
SP_OUTOFDISK	The disk is out of space.
SP_OUTOFMEMORY	The memory available for spooling is zero.

296.5 Cross-References

Escape(), ResetDC(), StartPage()

297 Escape

297.1 Synopsis

int Escape(HDC hdc, int nEscape, int cbInput, LPCSTR InData, void *OutData);

297.2 Description

The *Escape()* function provides access to a device functionality not accessible through GDI. The *hdc* parameter is the handle to the device-context. The *Escape* parameter identifies the escape function. The *InputSize* parameter is the size of the data block, in bytes, pointed to by the *InData* parameter. The *InData* parameter is the pointer to the input structure. *OutData* is the pointer to the output structure. If *OutData* is NULL, no data are returned.

297.3 Returns

If successful, the *Escape()* function returns a value greater than zero, except for QUERYESCSUPPORT. If the return value is zero, the escape function is not supported by this device or device driver.

297.4 Errors

If it is unsuccessful, the *Escape()* function returns a value less than zero or one of the following error codes:

SP_ERROR	This value indicates any error not described below.
SP_APPABORT	This value indicates that the current job was terminated.
SP_OUTOFDISK	This value indicates that the disk is out of space.
SP_OUTOFMEMORY	This value indicates that the memory available for spooling is zero.

297.5 Cross-References

None.

298 ExtDeviceMode

298.1 Synopsis

```
typedef struct tagDEVMODE {
          char        dmDeviceName[CCHDEVICENAME];
          UINT        dmSpecVersion;
          UINT        dmDriverVersion;
          UINT        dmSize;
          UINT        dmDriverExtra;
          DWORD       dmFields;
                 int  dmOrientation;
                 int  dmPaperSize;
                 int  dmPaperLength;
                 int  dmPaperWidth;
                 int  dmScale;
                 int  dmCopies;
                 int  dmDefaultSource;
                 int  dmPrintQuality;
                 int  dmColor;
                 int  dmDuplex;
                 int  dmYResolution;
```

```
        int         dmTTOption;
} DEVMODE;
int ExtDeviceMode(HWND Wnd, HANDLE hDriver, LPDEVMODE Output, LPSTR Device, LPSTR Port,
        LPDEVMODE Input, LPSTR Profile, WORD Mode);
```

298.2 Description

The *ExtDeviceMode()* function retrieves and modifies the device initialization information. This function uses the dialog box supplied by the driver to configure the printer driver. If the printer driver supports the *ExtDeviceMode()* it has to export its name. The *Wnd* parameter is the handle to the parent window of the dialog box. The *Driver* parameter is the handle to the device-driver module. The *Output* parameter is the pointer to the **DEVMODE** structure.

The driver writes the initialization information supplied in the *Input* parameter to this structure. *Device* is the pointer to the name of the printer device. It is a null-terminated string. *Port* is the pointer to the name of the port. *Input* is the pointer to the **DEVMODE** structure that supplies initialization information to the printer driver. *Profile* is the pointer to the name of the initialization file, where initialization information is recorded and read. If this parameter is NULL, default initialization information is used. *Mode* is a mask of values that determines the operation. If this parameter is zero, the *ExtDeviceMode()* function returns the number of bytes required by the printer driver's **DEVMODE** structure. Otherwise, the *Mode* parameter can be one or more of the following values:

DM_IN_BUFFER	This input value is used to prompt, copy, or update. It merges the printer driver's current print settings with the settings in the **DEVMODE** structure pointed to by the *Input* parameter; only the members indicated by the application in the *dmFields* member are updated; this value is also defined as DM_MODIFY.
DM_IN_PROMPT	This input value is used to show the printer driver's setup dialog box and change the settings in the printer's **DEVMODE** structure to values specified by the user; this value is also defined as DM_PROMPT.
DM_OUT_BUFFER	This output value changes the driver's current settings to the **DEVMODE** structure pointed to by the *Output* parameter; if this bit is zero, the *Output* parameter is NULL; this value is also defined as DM_COPY.
DM_OUT_DEFAULT	This output value updates the default settings usually stored in the initialization file, using the current contents of the printer driver's **DEVMODE** structure; this value is also defined as DM_UPDATE.

298.3 Returns

If *Mode* is zero, *ExtDeviceMode()* returns the size of the buffer that contains the printer driver initialization data. If *ExtDeviceMode()* displays the printer device setup dialog box, the return value is either IDOK or IDCANCEL, depending on which button is selected. If *ExtDeviceMode()* does not display the dialog box and the function is successful, the return value is IDOK. The return value is less than zero, if the function fails.

298.4 Errors

None.

298.5 Cross-References

CreateDC(), DeviceMode(), GetModuleHandle(), GetProcAddress(), LoadLibrary()

---

299 GetDeviceCaps

299.1 Synopsis

int GetDeviceCaps(HDC hdc, int InfoType);

299.2 Description

The *GetDeviceCaps()* function gets information about a specific device. The *hdc* parameter is a handle to a device-context. The value of the *InfoType* parameter determines the type of information retrieved by the function. The following values can be used for the *InfoType* parameter:

DRIVERVERSION	This value retrieves the device driver's version number.
TECHNOLOGY	This value retrieves device technology; one of the following values is returned:
DT_PLOTTER	The device is a vector plotter.
DT_RASDISPLAY	The device is a raster display.
DT_RASPRINTER	The device is a raster printer.
DT_RASCAMERA	The device is a raster camera.
DT_CHARSTREAM	The device is a character stream.
DT_METAFILE	The device is a metafile.
DT_DISPFILE	The device is a display file.
HORZSIZE	This value is the device’s physical width in millimeters.
VERTSIZE	This value is the device’s physical height in millimeters.
HORZRES	This value is the device’s width in pixels.
VERTRES	This value is the device’s height in pixels.
LOGPIXELSX	This value is the number of pixels per logical inch along the device’s width.
LOGPIXELSY	This value is the number of pixels per logical inch along the device’s height.
BITSPIXEL	This value is the number of adjacent color bits for each pixel.
PLANES	This value is the number of color planes.
NUMBRUSHES	This value is the number of device-specific brushes.
NUMPENS	This value is the number of device-specific pens.
NUMMARKERS	Number of device-specific markers.
NUMFONTS	This value is the number of device-specific fonts.
NUMCOLORS	This value is the number of entries in the device's color table.
ASPECTX	This value is the relative width of a device pixel that is used for line drawing.
ASPECTY	This value is the relative height of a device pixel that is used for line drawing.
ASPECTXY	This value is the diagonal width of a device pixel that is used for line drawing.
PDEVICESIZE	This value is the size, in bytes, of the PDEVICE internal structure.
CLIPCAPS	This value specifies the device's clipping capabilities. One of the following values will be returned:
CP_NONE	The output is clipped.
CP_RECTANGLE	The output is clipped to rectangles.
CP_REGION	The output is clipped to regions.

SIZEPALETTE	This value is the number of entries in the system palette. The return value is available only if the device driver is written for Windows 3.0 or later and sets the RC_PALETTE bit in the RASTERCAPS index.
NUMRESERVED	This value is the number of reserved entries in the system palette; the return value is available only if the device driver is written for Windows 3.0 or later and sets the RC_PALETTE bit in the RASTERCAPS index.
COLORRES	This value specifies the supported color resolution, in bits per pixel, of the device; the return value is available only if the device driver is written for Windows 3.0 or later and sets the RC_PALETTE bit in the RASTERCAPS index.
RASTERCAPS	This value specifies the device's raster capabilities; a combination of the following values are returned:
RC_BANDING	The device supports banding.
RC_BIGFONT	The device supports fonts larger than 64K.
RC_BITBLT	The device transfers bitmaps.
RC_BITMAP64	The device supports bitmaps larger than 64K.
RC_DEVBITS	The device supports device bitmaps.
RC_DI_BITMAP	The device supports the *SetDIBits()* and *GetDIBits()* functions.
RC_DIBTODEV	The device supports the *SetDIBitsToDevice()* function.
RC_FLOODFILL	The device performs flood fills.
RC_GDI20_OUTPUT	The device supports MS-Windows version 2.0 features.
RC_GDI20_STATE	There is a state block in the device-context.
RC_NONE	The device does not support raster operations.
RC_OP_DX_OUTPUT	The device supports dev opaque and DX array.
RC_PALETTE	The device is a palette-based.
RC_SAVEBITMAP	The device saves bitmaps locally.
RC_SCALING	The device supports scaling.
RC_STRETCHBLT	The device supports the *StretchBlt()* function.
RC_STRETCHDIB	The device supports the *StretchDIBits()* function.
CURVECAPS	The device's curve capabilities; a combination of the following values are returned:
CC_NONE	The device does not support curves.
CC_CIRCLES	The device supports circles.
CC_PIE	The device supports pie wedges.
CC_CHORD	The device supports chords.
CC_ELLIPSES	The device supports ellipses.
CC_WIDE	The device supports wide borders.
CC_STYLED	The device supports styled borders.
CC_WIDESTYLED	The device supports wide, styled borders.
CC_INTERIORS	The device supports interiors.
CC_ROUNDRECT	The device supports rectangles with rounded corners.

LINECAPS	The device's line capabilities; a combination of the following values are returned:
LC_NONE	The device does not support lines.
LC_POLYLINE	The device supports polylines.
LC_MARKER	The device supports markers.
LC_POLYMARKER	The device supports polymarkers.
LC_WIDE	The device supports wide lines.
LC_STYLED	The device supports styled lines.
LC_WIDESTYLED	The device supports wide, styled lines.
LC_INTERIORS	The device supports interiors.
POLYGONALCAPS	The device's polygonal capabilities; a combination of the following are returned:
PC_NONE	The device does not support polygons.
PC_POLYGON	The device supports alternate fill polygons.
PC_RECTANGLE	The device supports rectangles.
PC_WINDPOLYGON	The device supports winding fill polygons.
PC_SCANLINE	The device supports scan lines.
PC_WIDE	The device supports wide borders.
PC_STYLED	The device supports styled borders.
PC_WIDESTYLED	The device supports wide, styled borders.
PC_INTERIORS	The device supports interiors.
TEXTCAPS	The device's text capabilities; a combination of the following values are returned:
TC_OP_CHARACTER	The device supports character output precision; the device can place device fonts at any pixel location; this is a requirement for any device with device fonts.
TC_OP_STROKE	The device supports stroke output precision; the device can omit any stroke of a device font.
TC_CP_STROKE	The device supports stroke clip precision the device can clip device fonts to a pixel boundary.
TC_CR_90	The device supports 90-degree character rotation; the device can rotate characters only 90 degrees at a time.
TC_CR_ANY	The device supports character rotation at any angle; the device can rotate device fonts through any angle.
TC_SF_X_YINDEP	The device supports scaling independent of x and y directions; the device can scale device fonts separately in x and y directions.
TC_SA_DOUBLE	The device supports doubled characters for scaling; the device can double the size of device fonts.
TC_SA_INTEGER	The device supports integer multiples for scaling; the device can scale the size of device fonts in any integer multiple.
TC_SA_CONTIN	The device supports any multiples for exact scaling; the device can scale device fonts by any amount and still preserve the x and y ratios.

TC_EA_DOUBLE	The device supports double-weight; the device can make device fonts bold; if this bit is not set for printer drivers, an attempt is made to create bold device fonts by printing them twice.
TC_IA_ABLE	The device supports italics; the device can make device fonts italic; if this bit is not set, it is assumed that italics are not available.
TC_UA_ABLE	The device supports underlining, which indicates the device can underline device fonts; if this bit is not set, GDI creates underlines for device fonts.
TC_SO_ABLE	The device supports strikeouts; the device can strike out device fonts; if this bit is not set, strikeouts are created for device fonts.
TC_RA_ABLE	The device supports raster fonts. If this bit is set, raster and TrueType fonts available for this device are enumerated when the *EnumFonts()* or *EnumFontFamilies()* function are called; if this bit is not set, no raster or TrueType fonts are enumerated when the *EnumFonts()* or *EnumFontFamilies()* function are called.
TC_VA_ABLE	The device supports vector fonts. If this bit is set, the vector fonts available for this device are enumerated when the *EnumFonts()* or *EnumFontFamilies()* function are called; if this bit is not set, no Raster or TrueType fonts are enumerated when the *EnumFonts()* or *EnumFontFamilies()* function are called. This capability is only important for vector devices such as plotters that only can support Vector fonts; raster printer drivers and display drivers always numerate vector fonts.
TC_RESERVED	This value is reserved; it must be zero.

299.3 Returns

If the *GetDeviceCaps()* function is successful, it returns the requested information.

299.4 Errors

None.

299.5 Cross-References

EnumFonts(), EnumFontFamilies(), GetDIBits(), SetDIBits(), SetDIBitsToDevice(), StretchBlt(), StretchDIBits()

300 SetAbortProc, AbortProc

300.1 Synopsis

int SetAbortProc(HDC hdc, ABORTPROC AbortProcPtr);

BOOL CALLBACK AbortProc(HDC hdc, int error);

300.2 Description

The *SetAbortProc()* function installs the user-defined procedure, *AbortProc()*, that allows a print job to be canceled at the time of spooling. The *hdc* parameter is the handle to the printer device-context used for the current print job. The *AbortProcPtr* is the pointer to the *AbortProc()* function and is passed to the *SetAbortProc()* function as a second parameter.

The *AbortProc()* function is a user-defined, exported, printer callback function of type, ABORTPROC, whose address is passed in *SetAbortProc()*, called to cancel print job during spooling. The *hdc* parameter is the handle to the device-context and error specifies the current error. If error is zero there is no error. If error is SP_OUTOFDISK the application yields control using the *PeekMessage()* or *GetMessage()* parameter.

300.3 Returns

If the *AbortProc()* function returns TRUE, printing continues. If the *AbortDoc()* function returns FALSE, the print job is cancelled.

If successful, the *SetAbortProc()* function returns TRUE. Otherwise, it returns FALSE.

300.4 Errors

None.

300.5 Cross-References

GetMessage(), PeekMessage(), AbortDoc(), Escape()

---

301 SpoolFile

301.1 Synopsis

HANDLE SpoolFile(LPSTR Printer, LPSTR Port, LPSTR Job, LPSTR File);

301.2 Description

The *SpoolFile()* function places the *File* parameter in the spooler queue. All the parameters are null-terminated strings. *Printer* is the name of the printer. *Port* is the name of the local printer port. *Job* is the name of the job used by the spooler. This name has a maximum length of 32 bytes with the terminating zero. *File* is the name of the file to be used by the spooler.

301.3 Returns

If successful, the *SpoolFile()* function returns a valid global handle.

301.4 Errors

If unsuccessful, the *SpoolFile()* function returns one of the following error codes:

SP_ERROR	This code specifies any error not described below.
SP_NOTREPORTED	This error is not reported.
SP_APPABORT	The current job was terminated.
SP_USERABORT	The user canceled the print job.
SP_OUTOFDISK	The disk is out of space.
SP_OUTOFMEMORY	Memory available for spooling is zero.

301.5 Cross-References

Escape(), EndDoc()

---

302 StartDoc

302.1 Synopsis

```
typedef struct {
        int         Size;
        LPCSTR      DocName;
        LPCSTR      Output;
} DOCINFO;
```

int StartDoc(HDC hdc, DOCINFO *docinf);

302.2 Description

The *StartDoc()* function informs the output device about the beginning of the new print job. The *hdc* parameter is the handle to the device-context used for the current print job. The *docinf* parameter is a pointer to the **DOCINFO** structure.

Note: Do not use *StartDoc()* with metafiles.

The **DOCINFO** structure holds the filenames of the input file, the document, and the output file, if any. The members **DocName** and **Output** are pointers to NULL terminated strings. *DocName*'s maximum length is 32 bytes including the terminating NULL. If **Output** is NULL, the print job is directed to the printer. If any file name is specified, the print job is directed to that file.

302.3 Returns

If successful, the *StartDoc()* function returns a value greater than zero. If it is unsuccessful, the *StartDoc()* function returns SP_ERROR.

302.4 Errors

None.

302.5 Cross-References

None.

---

303 StartPage

303.1 Synopsis

int StartPage(HDC hdc);

303.2 Description

The *StartPage()* function informs the output device about the beginning of the new page. The *hdc* parameter is the handle to the device-context used for the current print job. This function disables *ResetDC()* until *EndPage()* is called.

303.3 Returns

If successful, the *StartPage()* function returns a value greater than or equal to zero. If it is not successful, the *StartPage()* function returns a value less than zero.

303.4 Errors

None.

303.5 Cross-References

Escape(), EndPage(), ResetDC()

---

304 QueryAbort

304.1 Synopsis

BOOL QueryAbort(HDC hdc, int reserved);

304.2 Description

The *QueryAbort()* function calls the *AbortProc()* callback function to determine whether printing should be terminated. The hdc parameter is the handle to the printer device-context used for the current print job, *reserved* is reserved parameter and must be zero.

304.2 Returns

If the *QueryAbort()* function returns TRUE, the print job should be terminated.

304.4 Errors

None.

304.5 Cross-References

AbortDoc(), AbortProc(), SetAbortProc()