Application Programming Interface for Windows

Volume 2
Section 4 - System Services
Section 5 - Application Support Functions
Application Programming Interface for Windows
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(), bitblt() 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 4 - System Services**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFreeSystemResources</td>
<td>1</td>
</tr>
<tr>
<td>SystemParametersInfo</td>
<td>1</td>
</tr>
<tr>
<td>GetWinFlags</td>
<td>6</td>
</tr>
<tr>
<td>GetSystemMetrics</td>
<td>7</td>
</tr>
<tr>
<td>GetVersion</td>
<td>8</td>
</tr>
<tr>
<td>SetTimer, TimerProc, KillTimer</td>
<td>9</td>
</tr>
<tr>
<td>SetDoubleClickTime, GetDoubleClickTime</td>
<td>9</td>
</tr>
<tr>
<td>GetTickCount, GetCurrentTime</td>
<td>10</td>
</tr>
<tr>
<td>LoadLibrary, FreeLibrary</td>
<td>11</td>
</tr>
<tr>
<td>LoadModule, FreeModule</td>
<td>11</td>
</tr>
<tr>
<td>ModuleFileName, GetModuleHandle, GetModuleUsage</td>
<td>13</td>
</tr>
<tr>
<td>GetProcAddress</td>
<td>13</td>
</tr>
<tr>
<td>MakeProcInstance, FreeProcInstance</td>
<td>14</td>
</tr>
<tr>
<td>LibMain</td>
<td>14</td>
</tr>
<tr>
<td>WEP</td>
<td>15</td>
</tr>
<tr>
<td>GetInstanceData</td>
<td>15</td>
</tr>
<tr>
<td>GetFreeSpace</td>
<td>16</td>
</tr>
<tr>
<td>GlobalAlloc, GlobalFree</td>
<td>16</td>
</tr>
<tr>
<td>GlobalCompact, LocalCompact</td>
<td>17</td>
</tr>
<tr>
<td>GlobalFix, GlobalUnfix</td>
<td>17</td>
</tr>
<tr>
<td>GlobalFlags, LocalFlags</td>
<td>18</td>
</tr>
<tr>
<td>GlobalHandle, LocalHandle</td>
<td>18</td>
</tr>
<tr>
<td>GlobalLock, GlobalUnlock</td>
<td>19</td>
</tr>
<tr>
<td>GlobalLRUNewest, GlobalLRUOldest</td>
<td>19</td>
</tr>
<tr>
<td>GlobalNotify, NotifyProc</td>
<td>20</td>
</tr>
<tr>
<td>GlobalReAlloc</td>
<td>20</td>
</tr>
<tr>
<td>GlobalSize, LocalSize</td>
<td>21</td>
</tr>
<tr>
<td>LocalInit, LocalShrink</td>
<td>21</td>
</tr>
<tr>
<td>Yield, DirectedYield</td>
<td>22</td>
</tr>
<tr>
<td>GetCurrentTask</td>
<td>23</td>
</tr>
<tr>
<td>GetNumTasks</td>
<td>23</td>
</tr>
<tr>
<td>GetWindowTask</td>
<td>23</td>
</tr>
<tr>
<td>IsTask</td>
<td>24</td>
</tr>
<tr>
<td>WinHelp</td>
<td>24</td>
</tr>
<tr>
<td>EnumTaskWindows, EnumTaskWndProc</td>
<td>25</td>
</tr>
<tr>
<td>WinExec</td>
<td>26</td>
</tr>
<tr>
<td>WinMain</td>
<td>26</td>
</tr>
<tr>
<td>ExitWindows</td>
<td>27</td>
</tr>
<tr>
<td>GetAsyncKeyState</td>
<td>27</td>
</tr>
<tr>
<td>qGetInputState</td>
<td>28</td>
</tr>
</tbody>
</table>
347 GetKeyboardState, SetKeyboardState 28
348 GetKeyNameText 29
349 GetKeyState 29
350 GetKBCodePage 30
351 OemKeyScan 30
352 MapVirtualKey 31
353 VkKeyScan 31
354 SwapMouseButton 32
355 GetKeyboardType 32
356 FindResource 33
357 LoadResource, FreeResource 34
358 LockResource 35
359 LoadString 35
360 LoadIcon 35
361 LoadBitmap 36
362 SetResourceHandler, LoadProc 37
363 SizeofResource 38
364 LoadMenu 38
365 LoadMenuIndirect 38
366 LoadAccelerators 39
367 AllocResource 39
368 BuildCommDCB 39
369 ClearCommBreak, SetCommBreak 40
370 CloseComm, OpenComm 40
371 EnableCommNotification 41
372 EscapeCommFunction 42
373 FlushComm 43
374 GetCommError 44
375 GetCommEventMask, SetCommEventMask 45
376 GetCommState, SetCommState 46
377 ReadComm, WriteComm 47
378 TransmitCommChar, UngetCommChar 47
379 GetDriveType 48
380 GetSystemDirectory 48
381 GetTempDrive 49
382 GetTempFileName 49
383 GetWindowsDirectory 50
384 OpenFile 50
385 SetHandleCount 52
386 _lclose 52
387 _lread 52
388 _lcreat 53
389 _llseek 53
390 _lopen 54
Section 5 - Application Support Functions

403 ExtractIcon
404 FindExecutable
405 GetPrivateProfileString, GetProfileString
406 WritePrivateProfileString, WriteProfileString
407 GetPrivateProfileInt, GetProfileInt
408 AnsiLower, AnsiLowerBuff
409 AnsiUpper, AnsiUpperBuff
410 AnsiNext, AnsiPrev
411 IsCharAlpha
412 IsCharAlphaNumeric
413 IsCharLower
414 IsCharUpper
415 lstrcmp, lstrcmpi
416 lstrcat, lstrcpy, lstrcpyn
417 lstrlen
418 wsprintf, wvprintf
419 IsDBCSLeadByte
420 ToAscii
421 AnsiToOem, AnsiToOemBuff
422 OemToAnsi, OemToAnsiBuff
423 CopyRect, SetRect, SetRectEmpty, InflateRect, OffsetRect
424 EqualRect, IsRectEmpty, PtInRect
425 IntersectRect, UnionRect, SubtractRect
426 OutputDebugString
427 DebugOutput
428 FatalAppExit
429 FatalExit
430 QuerySendMessage
431 LockInput
432 FlashWindow
433 MessageBeep
434 MessageBox 74
435 SetErrorMode 76
436 GetExpandedName 76
437 ChooseColor 77
438 ChooseFont 78
439 FindText, ReplaceText 80
440 GetOpenFileName, GetSaveFileName 82
441 GetFileTitle 85
442 PrintDlg 85
443 CommDlgExtendedError 86
444 MulDiv 87
Section 4 - System Services

305 GetFreeSystemResources

305.1 Synopsis
UINT GetFreeSystemResources(UINT ResourceType)

305.2 Description
The GetFreeSystemResources() function determines the percentage of free space available for all system resources or a system resource of a specific type. An application should not use this function to determine if it is possible to create a new resource object.

The resource type is specified in the ResourceType parameter and can be one of the following defined values:
- GFSR_SYSTEMRESOURCES: This value specifies all system resources.
- GFSR_USERRESOURCES: This value specifies USER resources; window and menu handles are considered USER resources.
- GFSR_GDIRESOURCES: This value specifies GDI resources; device-context handles, brushes, pens, regions, fonts, and bitmaps are considered GDI resources.

305.3 Returns
The GetFreeSystemResources() function returns the percentage of free space available for the system resource indicated.

305.4 Errors
None.

305.5 Cross-References
None.

306 SystemParametersInfo

306.1 Synopsis
BOOL SystemParametersInfo(UINT Operation, UINT Data1, void *Data2, UINT UpdateFlag);

306.2 Description
The SystemParametersInfo() function gets or sets a specific type of system information. The type of system information and the operation performed on that information is specified in the function's Operation parameter. The function's Data1 and Data2 parameters contain data unique to the operation being performed.

If the operation sets system information, the function uses the UpdateFlag parameter to determine if the change to the system information should be saved to the WIN.INI file. The value of the UpdateFlag parameter can be zero to indicate that the WIN.INI file should not be updated or it can be one or more of the following values OR'ed together:
- SPIF_UPDATEINIFILE: This value updates the WIN.INI file.
- SPIF_SENDWININICHANGE: This value broadcasts the WM_WININICHANGE message to all top-level windows; valid only when used in combination with the SPIF_UPDATEINIFILE value.

The following table contains the allowable values for the Command parameter and a description of its function:
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Beep Sound</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETBEEP</td>
<td>This value determines if the warning beep is set to on or off.</td>
</tr>
<tr>
<td>SPI_SETBEEP</td>
<td>This value sets the warning beep to on or off.</td>
</tr>
<tr>
<td><strong>Window Border</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETBORDER</td>
<td>This value gets the border multiplying factor that is used when</td>
</tr>
<tr>
<td></td>
<td>calculating the width of a window's sizing border.</td>
</tr>
<tr>
<td>SPI_SETBORDER</td>
<td>This value sets the border multiplying factor that is used when</td>
</tr>
<tr>
<td></td>
<td>calculating the width of a window's sizing border.</td>
</tr>
<tr>
<td><strong>Task Switching</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETFASTTASKSWITCH</td>
<td>This value determines if the fast task switching option is set on</td>
</tr>
<tr>
<td></td>
<td>or off.</td>
</tr>
<tr>
<td>SPI_SETFASTTASKSWITCH</td>
<td>This value turns the fast task switching option on or off.</td>
</tr>
<tr>
<td><strong>Desktop</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETGRIDGRANULARITY</td>
<td>This value gets the granularity value for the desktop's sizing</td>
</tr>
<tr>
<td></td>
<td>grid.</td>
</tr>
<tr>
<td>SPI_SETGRIDGRANULARITY</td>
<td>This value sets the granularity value for the desktop's sizing</td>
</tr>
<tr>
<td></td>
<td>grid.</td>
</tr>
<tr>
<td>SPI_SETDESKPATTERN</td>
<td>This value sets the desktop pattern.</td>
</tr>
<tr>
<td>SPI_SETDESKWALLPAPER</td>
<td>This value sets the bitmap used for the desktop wallpaper.</td>
</tr>
<tr>
<td><strong>Icons</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETICONTITLELOGFONT</td>
<td>This value gets the font information for the current font used to</td>
</tr>
<tr>
<td></td>
<td>draw icon titles.</td>
</tr>
<tr>
<td>SPI_SETICONTITLELOGFONT</td>
<td>This value sets the font that is used for drawing icon titles.</td>
</tr>
<tr>
<td>SPI_GETICONTITLEWRAP</td>
<td>This value determines if icon title word wrapping is set to on or</td>
</tr>
<tr>
<td></td>
<td>off.</td>
</tr>
<tr>
<td>SPI_SETICONTITLEWRAP</td>
<td>This value sets the icon title word wrapping option on or off.</td>
</tr>
<tr>
<td>SPI_ICONHORIZONTALSPACING</td>
<td>This value sets the number of pixels in an icon's cell width.</td>
</tr>
<tr>
<td>SPI_ICONVERTICALSPACING</td>
<td>This value sets the number of pixels in an icon's cell height.</td>
</tr>
<tr>
<td><strong>Keyboard</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETKEYBOARDDELAY</td>
<td>This value gets the keyboard's repeat-delay value</td>
</tr>
<tr>
<td>SPI_SETKEYBOARDDELAY</td>
<td>This value sets the keyboard's repeat-delay value</td>
</tr>
<tr>
<td>SPI_GETKEYBOARDSPEED</td>
<td>This value gets the keyboard's repeat-speed value.</td>
</tr>
<tr>
<td>SPI_SETKEYBOARDSPEED</td>
<td>This value sets the keyboard's repeat-speed value.</td>
</tr>
<tr>
<td><strong>Menus</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>SPI_GETMENUDROPALIGNMENT</td>
<td>This value determines if pop-up menus are aligned to the left or</td>
</tr>
<tr>
<td></td>
<td>right of its menu-bar item. Sets how pop-up menus are aligned relative to its menu-bar item.</td>
</tr>
<tr>
<td>SPI_SETMENUDROPALIGNMENT</td>
<td>This value sets how pop-up menus are aligned relative to its menu-bar item.</td>
</tr>
<tr>
<td><strong>Mouse</strong></td>
<td>Gets the mouse speed and the mouse threshold values.</td>
</tr>
<tr>
<td>SPI_GETMOUSE</td>
<td>This value gets the mouse speed and the mouse threshold values.</td>
</tr>
</tbody>
</table>
### SPI_SETMOUSE
This value sets the mouse speed and the mouse threshold values.

### SPI_SETDOUBLECLKHEIGHT
This value sets the height of the rectangle within which the second click of the mouse button double-click must fall for it to be registered as a mouse double-click.

### SPI_SETDOUBLECLICKTIME
This value sets the maximum number of milliseconds that can occur between the first and second mouse button clicks of a mouse button double-click.

### SPI_SETDOUBLECLKWIDTH
This value sets the width of the rectangle within which the second click of a double-click.

### SPI_SETMOUSEBUTTONSWAP
This value sets the meaning of the left and right mouse buttons.

### Screen Saver Value

<table>
<thead>
<tr>
<th>Command</th>
<th>Data1</th>
<th>Data2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_GETSCREENSAVEACTIVE</td>
<td>Value</td>
<td>This value determines if screen saving is set to on or off.</td>
</tr>
<tr>
<td>SPI_SETSCREENSAVEACTIVE</td>
<td>Value</td>
<td>This value sets the screen saver to on or off.</td>
</tr>
<tr>
<td>SPI_GETSCREENSAVETIMEOUT</td>
<td>Value</td>
<td>This value retrieves the screen saver's time-out setting.</td>
</tr>
<tr>
<td>SPI_SETSCREENSAVETIMEOUT</td>
<td>Value</td>
<td>This value sets the screen saver's time-out setting.</td>
</tr>
<tr>
<td>Language</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>SPI_LANGDRIVER</td>
<td>Value</td>
<td>This value forces the use of a new language driver.</td>
</tr>
</tbody>
</table>

The following table describes the use of the Data1 and Data2 parameters for each of the Command parameter's values:

<table>
<thead>
<tr>
<th>Command</th>
<th>Data1</th>
<th>Data2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_GETBEEP</td>
<td>Ignored.</td>
<td>This value is a pointer to a BOOL variable. The value of the variable is set to TRUE if the warning beep is on or FALSE if it is off.</td>
</tr>
<tr>
<td>SPI_GETBORDER</td>
<td>Ignored.</td>
<td>This value is a pointer to an integer variable. The value of the border multiplying factor is assigned to the variable.</td>
</tr>
<tr>
<td>SPI_GETFASTTASKSWITCH</td>
<td>Ignored.</td>
<td>This value is a pointer to a BOOL variable. The value of the variable is set to TRUE if fast task switching is on or FALSE if it is off.</td>
</tr>
<tr>
<td>SPI_GETGRIDGRANULARITY</td>
<td>Ignored.</td>
<td>This value is a pointer to an integer variable. The value of the grid-granularity setting is assigned to the variable.</td>
</tr>
<tr>
<td>SPI_GETICONTITLELOGFONT</td>
<td>The size of the LOGFONT structure pointed to by the Data2 parameter.</td>
<td>This value is a pointer to a LOGFONT structure that is assigned the is filled with logical-font information.</td>
</tr>
<tr>
<td>SPI_GETICONTITLEWRAP</td>
<td>Ignored.</td>
<td>This value is a pointer to a BOOL variable. The value of the variable is set to TRUE if icon title wrapping is on or FALSE if it is off.</td>
</tr>
<tr>
<td>SPI_GETKEYBOARDDELAY</td>
<td>Ignored.</td>
<td>This value is a pointer to an integer variable. The value of the keyboard repeat-delay setting is assigned to the variable.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SPI_GETKEYBOARD-SPEED</td>
<td>This value is a pointer to a WORD variable. The value of the keyboard repeat-speed setting is assigned to the variable.</td>
<td></td>
</tr>
<tr>
<td>SPI_GETMENUDROP-ALIGNMENT</td>
<td>This value is a pointer to a BOOL variable. The value of the variable is set to TRUE if pop-up menus are right-aligned or FALSE if they are left-aligned.</td>
<td></td>
</tr>
<tr>
<td>SPI_GETMOUSE</td>
<td>This value is a pointer to an array of three integers. The first integer is assigned the value of the MouseThreshold1 WIN.INI entry. The second integer is assigned the value of the MouseThreshold2 WIN.INI entry. The third integer is assigned the value of the MouseSpeed WIN.INI entry.</td>
<td></td>
</tr>
<tr>
<td>SPI_GETSCREENSAVERACTIVE</td>
<td>This value is a pointer to a BOOL variable. The value of the variable is set to TRUE if the screen saver is active or FALSE if it is not active.</td>
<td></td>
</tr>
<tr>
<td>SPI_GETSCREENSAVETIMEOUT</td>
<td>This value is a pointer to an integer variable. The value, in milliseconds, of the screen saver's time-out is assigned to the variable.</td>
<td></td>
</tr>
<tr>
<td>SPI_ICONHORIZONTALSPACING</td>
<td>If the value of the Data2 parameter is NULL, this parameter's value should be the new width, in pixels, for the horizontal spacing of icons. If the value of the Data2 parameter is not NULL, this parameter is ignored.</td>
<td>If this value is a pointer to an integer variable, the value of the current icon horizontal spacing is returned in the variable. If this value is NULL, the value in the Data1 parameter is used to set the horizontal spacing of icons.</td>
</tr>
<tr>
<td>SPI_ICONVERTICALSPACING</td>
<td>If the value of the Data2 parameter is NULL, this parameter's value should be the new height, in pixels, for the vertical spacing of icons. If the value of the Data2 parameter is not NULL, this parameter is ignored.</td>
<td>If this value is a pointer to an integer variable, the value of the current icon vertical spacing is returned in the variable. If this value is NULL, the value in the Data1 parameter is used to set the vertical spacing of icons.</td>
</tr>
<tr>
<td>SPI_LANGDRIVER</td>
<td>This value is a pointer to a string containing the file-name of the new language driver.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETBEEP</td>
<td>TRUE turns the warning beep on. FALSE turns the warning beep off.</td>
<td>Ignored.</td>
</tr>
<tr>
<td>SPI_SETBORDER</td>
<td>The new value of the window border multiplying factor.</td>
<td>Ignored.</td>
</tr>
<tr>
<td>SPI_SETDESKTOP-PATTERN</td>
<td>If the value of the Data2 parameter is NULL, this parameter's value should be -1. Otherwise, this parameter is ignored.</td>
<td>If this value is NULL and the value of the Data1 parameter is -1, the value of WIN.INI file's desktop pattern is reread. If this value is not NULL, it is assumed to be a pointer to a null-terminated string. The string should contain eight RGB values representing the desktop pattern.</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SPI_SETDESKWALLPAPER</td>
<td>Ignored. This value is a pointer to a string. The string contains the name of a bitmap file to be used for the desktop wallpaper.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETDOUBLECLKHEIGHT</td>
<td>Number of pixels to use for the mouse button's double-click height.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETDOUBLECLICKTIME</td>
<td>Number of milliseconds to use for the mouse button's double-click time.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETDOUBLECLKWIDTH</td>
<td>Number of pixels to use for the mouse button's double-click width.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETFASTTASKSWITCH</td>
<td>TRUE turns the fast task switching on. FALSE turns the fast task switching off.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETGRIDGRANULARITY</td>
<td>Number of pixels to use for grid granularity.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETICONTITLELOGFONT</td>
<td>If the value of the Data2 parameter is NULL, this parameter's value should be zero. Otherwise, the parameter should contain the size of the LOGFONT structure pointed to by the Data2 parameter. If the value of the Data1 parameter is zero and this parameter is set to NULL, the font information that was in effect when the session was started is used to draw icon titles. Other, this parameter is a pointer to a LOGFONT structure that defines a logical-font to use when drawing an icon's title.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETICONTITLEWRAP</td>
<td>TRUE turns the icon title wrapping feature on. FALSE turns the icon title wrapping feature off.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETKEYBOARDDELAY</td>
<td>The new value for the keyboard's delay setting.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETKEYBOARDSPEED</td>
<td>The new value for the keyboard's repeat-speed setting.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETMENUDROPALIGNMENT</td>
<td>TRUE sets drop-down menus to be right-aligned. FALSE sets drop-down menus to be left-aligned.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETMOUSE</td>
<td>Ignored. This value is a pointer to an array of three integers. The MouseThreshold1 WIN.INI entry is set the value of the first integer. The MouseThreshold2 WIN.INI entry is set the value of the first integer. The MouseSpeed WIN.INI entry is set the value of the third integer.</td>
<td></td>
</tr>
<tr>
<td>SPI_SETMOUSEBUTTONSWAP</td>
<td>TRUE sets the right mouse button to act as the left mouse button and left mouse button to act as</td>
<td></td>
</tr>
</tbody>
</table>
the right mouse button.
FALSE restores the mouse buttons to their normal meanings.

<table>
<thead>
<tr>
<th>SPI_SETSCREENSAVEACTIVE</th>
<th>TRUE activates the screen saving feature.</th>
<th>Ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FALSE deactivates the screen saving feature.</td>
<td></td>
</tr>
</tbody>
</table>

| SPI_SETSCREENSAVETIMEOUT | The new value, in seconds, for the screen saver's idle time-out. | Ignored. |

306.3 Returns
If the SystemParametersInfo() function is successful, it returns TRUE. Otherwise, it returns FALSE.

306.4 Errors
None.

306.5 Cross-References
None.

307 GetWinFlags

307.1 Synopsis
DWORD GetWinFlags(void);

307.2 Description
The GetWinFlags() function gets the system and memory configuration.

307.3 Returns
The GetWinFlags() function's return value can be a combination of the following values described below:

- WF_80x87: The system contains a Intel math coprocessor.
- WF_CPU286: The system contains a Intel 80286 or equivalent CPU.
- WF_CPU386: The system contains a Intel 80386 or equivalent CPU.
- WF_CPU486: The system contains a Intel 80486 or equivalent CPU.
- WF_ENHANCED: Windows is running in 386-enhanced mode; if the WF_ENHANCED is set, the WF_PMODE flag is also set.
- WF_WIN386: Same as WF_ENHANCED.
- WF_STANDARD: Windows is running in standard mode; if the WF_STANDARD is set, the WF_PMODE flag is also set.
- WF_WIN286: Same as WF_STANDARD.
- WF_PMODE: Windows is running in protected mode; this flag is always set.
- WF_PAGING: Windows is running on a system with paged memory.

307.4 Errors
None.

307.5 Cross-References
None.
308 GetSystemMetrics

308.1 Synopsis

    int GetSystemMetrics (int InfoType);

308.2 Description

The GetSystemMetrics() function retrieves information about the width and height, in pixels, of the various elements displayed by the system and also retrieves some other miscellaneous system information. The InfoType parameter specifies the type of information that is desired. The InfoType parameter can be one of the following values:

- **SM_CXBORDER**: This value specifies the frame width of a window that cannot be sized.
- **SM_CYBORDER**: This value specifies the frame height of a window that cannot be sized.
- **SM_CYCAPTION**: This value specifies the window title height; this is the title height plus the height of the window frame that cannot be sized (SM_CYBORDER).
- **SM_CXCURSOR**: This value specifies the current cursor width.
- **SM_CYCURSOR**: This value specifies the current cursor height.
- **SM_CXDOUBLECLK**: This value specifies the width of the rectangle around the location of the first mouse button click in a mouse button double-click sequence; the second mouse button click must occur within this rectangle for the system to consider the two mouse button clicks a button double-click.
- **SM_CYDOUBLECLK**: This value specifies the height of the rectangle around the location of the first mouse button click in a mouse button double-click sequence; the second mouse button click must occur within this rectangle for the system to consider the two mouse button clicks a button double-click.
- **SM_CXDLGFRAME**: This value specifies the frame width of the window when the window has the WS_DLGFRAME style.
- **SM_CYDLGFRAME**: This value specifies the frame height of the window when the window has the WS_DLGFRAME style.
- **SM_CXFRAME**: This value specifies the frame width of the window that can be sized.
- **SM_CYFRAME**: This value specifies the frame height of the window that can be sized.
- **SM_CXFULLSCREEN**: This value specifies the width of a window client area for a full-screen window.
- **SM_CYFULLSCREEN**: This value specifies the height of a window client area for a full-screen window (the same value as the height of the screen minus the height of the window title).
- **SM_CXICON**: This value specifies the icon width.
- **SM_CYICON**: This value specifies the icon height.
- **SM_CXICONSPACING**: This value specifies the rectangle width used by the system to position tiled icons.
- **SM_CYICONSPACING**: This value specifies the rectangle height used by the system to position tiled icons.
- **SM_CYKANJIWINDOW**: This value specifies the Kanji window height.
- **SM_CYMENU**: This value specifies the single-line menu bar height; this value is the menu height minus the window frame height that cannot be sized (SM_CYBORDER).
- **SM_CXMIN**: This value specifies the minimum window width.
- **SM_CYMINTop**: This value specifies the minimum window height.
SM_CXMINTRACK This value specifies the minimum tracking window width.
SM_CYMINTRACK This value specifies the minimum tracking window height.
SM_CXSCREEN This value specifies the screen width.
SM_CYSCREEN This value specifies the screen height.
SM_CXHSCROLL This value specifies the arrow bitmap width on a horizontal scroll bar.
SM_CYHSCROLL This value specifies the arrow bitmap height on a horizontal scroll bar.
SM_CXVSCROLL This value specifies the arrow bitmap width on a vertical scroll bar.
SM_CYVSCROLL This value specifies the arrow bitmap height on a vertical scroll bar.
SM_CXSIZE This value specifies the width of bitmaps contained in the title bar.
SM_CYSIZE This value specifies the height of bitmaps contained in the title bar.
SM_CXHTHUMB This value specifies the thumb height on the vertical scroll bar.
SM_DBCSENABLED A non-zero value is returned if double-byte characters are being used; zero is returned if double-byte characters are not being used.
SM_DEBUG A non-zero value is returned if a debug version of the system is being used; zero is returned if a debug version of the system is not being used.
SM_MENUDROPALIGNMENT This value specifies the current alignment of pop-up menus to their respective menu item. Zero is returned when pop-up menus are aligned to the left of its menu-bar item; a non-zero value is returned when pop-up menus are aligned to the right of its menu-bar item.
SM_MOUSEPRESENT Zero is returned when a mouse is not installed on the system; a non-zero value is returned when a mouse is installed on the system.
SM_PENWINDOWS If Pen Windows is installed, a handle to the Pen Windows dynamic-link library (DLL) is returned.
SM_SWAPBUTTON Zero is returned when the left and right mouse buttons are not swapped; a non-zero value is returned when the left and right mouse buttons are swapped.

### 308.3 Returns
If the `GetSystemMetrics()` function is successful, the requested information is returned.

### 308.4 Errors
None.

### 308.5 Cross-References
`GetWinFlag()`

---

### 309 GetVersion

#### 309.1 Synopsis
```c
DWORD GetVersion(void);
```

#### 309.2 Description
The `GetVersion()` function retrieves the current versions of both Windows and MS-DOS.

#### 309.3 Returns
If successful, the `GetVersion()` function returns the Windows version number in the low-order word of the return value and the MS-DOS version number in the high-order word. The high-order byte in each word contains the major version number and the low-order byte contains the minor version number.
309.4 Errors
None.

309.5 Cross-References
None.

310 SetTimer, TimerProc, KillTimer

310.1 Synopsis
UINT SetTimer(HWND hWnd, UINT TimerID, UINT Notify, TIMERPROC TimerProc);
void CALLBACK TimerProc(HWND hWnd, UINT msg, UINT TimerID, DWORD Time);
BOOL KillTimer(HWND hWnd, UINT TimerID);

310.2 Description
The SetTimer() function creates a new system timer. The TimerID parameter is the identifier associated with the new timer. If the hWnd parameter is NULL, the TimerID parameter is ignored. The Notify parameter's value defines at what interval, in milliseconds, the application is sent a WM_TIMER message. If the value of the TimerProc parameter is NULL, the WM_TIMER message is posted to the message queue of the window given in the hWnd parameter. Otherwise, the WM_TIMER is sent to the procedure given in the TimerProc parameter. The TimerProc parameter contains a procedure-instance address of a TIMERPROC callback function whose name has been exported in the application's module-definition file.

TimerProc() is an application-defined callback function that processes WM_TIMER messages. The hWnd parameter contains the handle of a window associated with the timer. The msg parameter contains the value WM_TIMER. The TimerID parameter contains the identifier of the system timer. The Time parameter contains the current system time.

KillTimer() removes a system timer. The TimerID parameter is the identifier of the timer to be removed. The hWnd parameter is the handle of the window used when the timer was created by the SetTimer() function. When a timer is removed, any unprocessed WM_TIMER messages for the timer are removed from the associated window's message queue.

310.3 Returns
If the SetTimer() function is successful and the value of its hWnd parameter is NULL, it returns the new timer's identifier. If the SetTimer() function is successful and the value of its hWnd parameter is not NULL, it returns a non-zero value. If the SetTimer() function is not successful, it returns zero.

TimerProc() does not return a value.

If KillTimer() is successful, it returns TRUE. Otherwise, it returns FALSE.

310.4 Errors
None.

310.5 Cross-References
None.

311 SetDoubleClickTime, GetDoubleClickTime

311.1 Synopsis
void SetDoubleClickTime(UINT Time);
UINT GetDoubleClickTime(void);

311.2 Description
The SetDoubleClickTime() function sets the maximum number of milliseconds that can occur between the first and second mouse button clicks of a mouse button double-click. The Time parameter contains the maximum number of milliseconds allowed.
GetDoubleClickTime() returns the maximum number of milliseconds that can occur between the first and second mouse button clicks of a mouse button double-click.

311.3 Returns

SetDoubleClickTime() does not return a value.

GetDoubleClickTime() returns the maximum number of milliseconds that can occur between the first and second mouse button clicks of a mouse button double-click.

311.4 Errors

None.

311.5 Cross-References

None.

312 GetTickCount, GetCurrentTime

312.1 Synopsis

DWORD GetTickCount(void);

DWORD GetCurrentTime(void);

312.2 Description

The GetTickCount() function returns the number of milliseconds that have elapsed since the session started. If the session is run for approximately 49 days, the tick count value rolls back over to zero.

GetCurrentTime() is identical to the GetTickCount() function.

312.3 Returns

When GetTickCount() and GetCurrentTime() are successful, they return the number of milliseconds that have elapsed since the session started.

312.4 Errors

None.

312.5 Cross-References

None.

313 GetTimerResolution

313.1 Synopsis

DWORD GetTimerResolution(void);

313.2 Description

The GetTimerResolution() function returns the number of microseconds for each timer tick.

313.3 Returns

GetTimerResolution() returns the number of microseconds for each timer tick.

313.4 Errors

None.

313.5 Cross-References

None.
314 LoadLibrary, FreeLibrary

314.1 Synopsis

HINSTANCE LoadLibrary(LPCSTR lpszFileName);

void FreeLibrary(HINSTANCE hInst);

314.2 Description

The LoadLibrary() function is used to load a library module. The file name used with the function is specified in lpszFileName parameter.

If the lpszFileName string does not contain the full path, the following directories are searched:

- the current directory
- the Windows directory (retrieved by GetWindowsDirectory())
- the system directory (retrieved by GetSystemDirectory())
- the directory containing the executable file for the current task (retrieved by GetModuleFileName())
- the directories listed in the PATH environment variable
- the directories mapped in the network

If the library module to be loaded already resides in memory, the LoadLibrary() function increases the reference count of the module.

FreeLibrary() decreases by 1 the reference count of the module identified by the hInst parameter. If the reference count is decremented to zero, the module is removed, and all the memory associated with it is freed.

314.3 Returns

LoadLibrary() returns the instance handle of the loaded library module, if it is successful. If the function fails, it returns the value less than HINSTANCE_ERROR.

FreeLibrary() does not return a value.

314.4 Errors

The cause of failure of the LoadLibrary() function and the error codes can be:

0 There is insufficient memory to load the module or corrupted library file.
2 The file is not found.
4 The path is not found.
5 A sharing or network-protection error has occurred.
8 There is insufficient memory to start the application.
11 A library file is invalid.
14 The type of library file is unknown.
19 The file is compressed.
20 The DLL file is invalid or one of the DLLs it requires is corrupt.
21 The library module needs 32-bit extensions not provided by the system.

314.5 Cross-References

LoadModule(), FreeModule(), WinExec()

315 LoadModule, FreeModule

315.1 Synopsis

typedef struct tagLOADPARAMS {
    WORD    segEnv;
    LPSTR   lpszCmdLine;
} LOADPARAMS;
LPUINT lpShow;
LPUINT lpReserved;
}

HINSTANCE LoadModule(LPCSTR lpszFileName, LPVOID lpvParamBlock);
BOOL FreeModule(HINSTANCE hInst);

315.2 Description
The `LoadModule()` function loads and executes an application module. The name of the application module is provided in `lpszFileName` parameter. The `lpvParamBlock` parameter is a pointer to a LOADPARAMS structure.

The `segEnv` field in the LOADPARAMS structure contains the segment for the new application environment for the application being launched. If it is set to 0, the new application receives a copy of the parent application's environment block.

The `lpszCommandLine` parameter points to a null-terminated string (up to 120 characters long) that specifies the command line string for the application being launched. It points to an empty string when no command line is provided. It cannot be set to NULL.

The `lpShow` parameter is a pointer to an array of two UINT values. The first element of the array must be set to 2. The second element is the `nCmdShow` value that is passed to `ShowWindow()` when the main application window is being shown.

If the `lpszFileName` string does not contain the full path, the following directories are searched:
- the current directory
- the Windows directory (retrieved by `GetWindowsDirectory()`)
- the system directory (retrieved by `GetSystemDirectory()`)
- the directory containing the executable file for the current task (retrieved by `GetModuleFileName()`)
- the directories listed in the PATH environment variable
- the directories mapped in the network

If the module to be loaded already resides in memory, the `LoadModule()` function creates another instance of the application.

`FreeModule()` decreases by 1 the reference count of the module identified by the `hInst` parameter. If the reference count is decremented to zero, the module is removed and all the memory associated with it is freed.

315.3 Returns
`LoadModule()` returns the instance handle of the loaded module, if it is successful. It returns the value less than HINSTANCE_ERROR if it is unsuccessful.

`FreeModule()` returns TRUE if the module's memory has been freed. Otherwise, it returns FALSE.

315.4 Errors
The cause of failure of the `LoadLibrary()` function and the error codes can be as follows:
There is insufficient memory to load the module or corrupted executable file.
The file is not found.
The path is not found.
A sharing or network-protection error has occurred.
There is insufficient memory to start the application.
An invalid executable file was discovered.
An unknown executable file type was discovered.
A second attempt was made to load an executable file with multiple data segments not marked read-only.
The file is compressed.
A DLL file is invalid or one of the DLLs it requires is corrupt.
The library module needs 32-bit extensions not provided by the system.

315.5 Cross-References
LoadLibrary(), FreeLibrary(), WinExec()

316 GetModuleFileName, GetModuleHandle, GetModuleUsage

316.1 Synopsis
int GetModuleFileName(HINSTANCE hInst, LPSTR lpszFileName, int cbFileName);
HMODULE GetModuleHandle(LPCSTR lpszModule);
int GetModuleUsage(HINSTANCE hInst);

316.2 Description
GetModuleFileName(), GetModuleHandle(), and GetModuleUsage() are used to obtain information about a loaded module.

GetModuleFileName() retrieves the null-terminated filename, including the full path, of the file from which the module specified by hInst parameter has been loaded. The hInst parameter can be an instance handle or a module handle.
The lpszFileName parameter points to a buffer to which the filename is copied. The cbFileName parameter specifies the length of the buffer. If the filename is longer than the buffer, it is truncated.

GetModuleHandle() obtains a handle of the module specified by name in the lpszModule parameter.

GetModuleUsage() returns the reference count for the module specified by hInst parameter. The hInst parameter can be an instance handle or a module handle. The reference count of a module is increased by one by every call to LoadLibrary() or LoadModule() and decreased by one by calls to FreeModule() or FreeLibrary().

316.3 Returns
GetModuleFileName() returns the number of bytes copied to the buffer. Otherwise, it returns zero.
GetModuleHandle() returns the module handle. Otherwise, it returns zero. GetModuleUsage() returns the reference handle of the given module. Otherwise, it returns zero.

316.4 Errors
None.

316.5 Cross-References
LoadLibrary(), LoadModule(), FreeLibrary(), FreeModu

317 GetProcAddress

317.1 Synopsis
FARPROC GetProcAddress(HINSTANCE hInst, LPCSTR lpszProcName);
317.2 Description
The GetProcAddress() function retrieves the address of a function in the module specified in the hInst parameter.

The hInst parameter can be either an instance handle or a module handle.

The lpszProcName parameter specifies the function whose address must be obtained. If the function is to be searched by name, the lpszProcName parameter is a pointer to a null-terminated function name string. If the function is identified by its ordinal, the high-order word of lpszProcName must be zero and the low-order word must contain the ordinal value of the function.

317.3 Returns
If the requested function exists in the module, the function returns it's address. If the function could not be located, the return value is NULL.

317.4 Errors
None.

317.5 Cross-References
None.

318 MakeProcInstance, FreeProcInstance
318.1 Synopsis
FARPROC MakeProcInstance(FARPROC lpProc, HINSTANCE hInst);
void FreeProcInstance(FARPROC lpProc);

318.2 Description
The MakeProcInstance() function creates a procedure instance of a given function. The address of the function is specified in lpProc parameter.

The procedure instance binds an exported function to an instance data segment of the application identified by the hInst parameter, so when the function is called, it has access to the data in this segment. In this way multiple instances of an application can call the same function and the function is able to use instance-specific data.

Dynamic-link libraries (DLLs) cannot have multiple data instances, so MakeProcInstance() returns the address specified in lpProc parameter.

The procedure-instance address of the function should be used when passing the pointer of a callback function to the system (for example, window procedure, enumeration procedure, etc.).

FreeProcInstance() frees the procedure instance specified in lpProc parameter. After the procedure instance has been freed, it cannot be used to call the function.

318.3 Returns
The MakeProcInstance() function returns the procedure-instance address, if it is successful. Otherwise, it returns a NULL.

318.4 Errors
None.

318.5 Cross-References
None.

319 LibMain
319.1 Synopsis
int CALLBACK LibMain(HINSTANCE hInst, WORD wDataSeg, WORD wHeapSize,
LPSTR lpszCmdLine);
319.2 Description
The `LibMain()` function is an entry point of a dynamic-link library (DLL) and is called by the system at the time the DLL is loaded. Every DLL should contain an exported procedure with this name.

The `hInst` parameter is the instance handle of the DLL module, the `wDataSeg` parameter specifies the selector of the data segment of the DLL. The `wHeapSize` parameter provides the size of the local heap in that data segment. By the time the `LibMain()` function is called, the local heap has already been initialized. The `lpszCmdLine` parameter is a pointer to the command line string.

319.3 Returns
The function returns 1, if it is successful. Otherwise, it returns zero.

319.4 Errors
None.

319.5 Cross-References
`WEP()`

---

320 WEP

320.1 Synopsis
```c
int CALLBACK WEP(int nExitType);
```

320.2 Description
The `WEP()` (Windows Exit Procedure) function in a dynamic-link library (DLL), is called by the system at the time the DLL is unloaded. It performs whatever cleanup is needed. The DLL does not require an exported entry point.

The `nExitType` parameter specifies the type of exit that is taking place. It can be one of the following:

- `WEP_FREE_DLL` Only the given library task is being terminated.
- `WEP_SYSTEM_EXIT` The whole system is shutting down.

320.3 Returns
The function returns 1, if it is successful. Otherwise, it returns zero.

320.4 Errors
None.

320.5 Cross-References
`LibMain()`

---

321 GetInstanceData

321.1 Synopsis
```c
int GetInstanceData(HINSTANCE hndlinst, BYTE *npDataBuff, int nbData);
```

321.2 Description
`GetInstanceData()` makes a copy of the previous instance of an application into the data area of the current instance. The parameter `hndlinst` specifies a previous instance of the application which has to be copied. The parameter `npDataBuff` contains a pointer to the buffer that will contain the current instance. The parameter `nbData` identifies the number of bytes to be copied.

The function `GetInstanceData()` if successful returns the number of bytes copied. Otherwise, it returns zero.

321.4 Errors
None.
321.5 Cross-References
None.

322 GetFreeSpace

322.1 Synopsis
DWORD GetFreeSpace(UINT uFlags);

322.2 Description
The GetFreeSpace() function retrieves the number of bytes of free memory available. The uFlags parameter is currently ignored. This function is not applicable to virtual memory systems.

322.3 Returns
GetFreeSpace() returns the number of bytes of available memory, if it is successful.

322.4 Errors
None.

322.5 Cross-References
None.

323 GlobalAlloc, GlobalFree, LocalAlloc, LocalFree

323.1 Synopsis
HGLOBAL GlobalAlloc(UINT uFlags, DWORD dwBytes);
HANDLE GlobalFree(HANDLE hMem);
HLOCAL LocalAlloc(UINT uFlags, DWORD dwBytes);
HANDLE LocalFree(HANDLE hMem);

323.2 Description
The GlobalAlloc() function allocates dwBytes bytes and returns a handle to the memory segment that can be accessed via GlobalLock().

<table>
<thead>
<tr>
<th>uFlags Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHND</td>
<td>This value is equivalent to GMEM_MOVEABLE and GMEM_ZEROINIT.</td>
</tr>
<tr>
<td>GMEM_DDESHARE</td>
<td>This value is used for DDE only; equivalent to GMEM_SHARE.</td>
</tr>
<tr>
<td>GMEM_DISCARDABLE1</td>
<td>This value marks the segment as discardable; can only be used with</td>
</tr>
<tr>
<td></td>
<td>GMEM_MOVEABLE.</td>
</tr>
<tr>
<td>GMEM_FIXED2</td>
<td>This value marks the segment as fixed; GMEM_FIXED and GMEM_MOVEABLE are mutually exclusive.</td>
</tr>
<tr>
<td>GMEM_GPTR</td>
<td>This value is equivalent to GMEM_FIXED AND GMEM_ZEROINIT.</td>
</tr>
<tr>
<td>GMEM_LOWER1</td>
<td>This value is equivalent to GMEM_NOT_BANKED.</td>
</tr>
<tr>
<td>GMEM_MOVEABLE1</td>
<td>This value marks the segment as moveable; GMEM_FIXED and</td>
</tr>
<tr>
<td></td>
<td>GMEM_MOVEABLE are mutually exclusive.</td>
</tr>
<tr>
<td>GMEM_NOCOMPACT1</td>
<td>This value does not attempt to compact or discard memory.</td>
</tr>
<tr>
<td>GMEM_NODISCARD1</td>
<td>This value does not attempt to discard memory.</td>
</tr>
<tr>
<td>GMEM_NOT_BANKED2</td>
<td>This value marks the segment as non-banked; cannot be used with</td>
</tr>
<tr>
<td></td>
<td>GMEM_NOTIFY.</td>
</tr>
<tr>
<td>GMEM_NOTIFY1</td>
<td>This value calls the notification function if the segment is discarded.</td>
</tr>
<tr>
<td>GMEM_SHARE</td>
<td>This value marks the segment as shared, accessible by other applications.</td>
</tr>
</tbody>
</table>
GMEM_ZEROINIT
This value initializes the memory segment to zero.

GMEM_GPTR
This value is equivalent to GMEM_FIXED and GMEM_ZEROINIT.

1 These flags are ignored in VM environments. However, this should not affect the functionality of your program.

2 These flags are non-portable and are currently ignored. Your code should not depend on these flags.

GlobalFree() frees the memory segment specified by the handle hMem. GlobalFree() cannot free a locked memory segment or a memory segment with a lock count greater than zero. To find the number of locks on a memory segment, see the GlobalFlags() function. Once a memory segment is freed, the handle to that memory segment should not be used again.

The LocalAlloc() and LocalFree() functions call the GlobalAlloc() and GlobalFree() functions respectively.

323.3 Returns
GlobalAlloc() and LocalAlloc() return a handle of the newly allocated memory segment, if they are successful. If unsuccessful, both functions return zero.

GlobalFree() and LocalFree() return zero, if they are successful. GlobalFree() and LocalFree() also return zero, if the handle passed to it does not exist. If unsuccessful, GlobalFree() and LocalFree() return hMem.

323.4 Errors
None.

323.5 Cross-References
GlobalLock(), GlobalUnlock(), GlobalFlags()

324 GlobalCompact, LocalCompact

324.1 Synopsis
DWORD GlobalCompact(DWORD MinFree);
DWORD LocalCompact(DWORD MinFree);

324.2 Description
The GlobalCompact() function rearranges the memory content until MinFree bytes of memory can no longer be rearranged.

LocalCompact() calls GlobalCompact().

324.3 Returns
GlobalCompact() and LocalCompact() return the number of bytes available in the largest contiguous memory segment. If MinFree is zero, GlobalCompact() returns the number of bytes available in the largest contiguous memory segment if all discardable memory segments are removed.

GlobalCompact() and LocalCompact() currently do nothing and return 4194304 bytes.

324.4 Errors
None.

324.5 Cross-References
None.

325 GlobalFix, GlobalUnfix

325.1 Synopsis
void GlobalFix(HANDLE hMem);
void GlobalUnfix(HANDLE hMem);
325.2 Description
The GlobalFix() function prevents the global memory object specified in the hmem parameter from moving in linear memory.

The GlobalUnfix() function allows the global memory object specified in the hmem parameter to be moved in linear memory. GlobalFix() and GlobalUnfix() are unnecessary in the virtual memory environment, and therefore, currently perform no operation.

325.3 Returns
None.

325.4 Errors
None.

325.5 Cross-References
None.

326 GlobalFlags, LocalFlags

326.1 Synopsis
UINT GlobalFlags(HANDLE hMem);
UINT LocalFlags(HANDLE hMem);

326.2 Description
The GlobalFlags() function returns the flag associated with the global memory segment specified by the hMem parameter.

The LocalFlags() function calls GlobalFlags().

326.3 Returns
The functions return the flags and lock count of the specified memory segment, if they are successful. The flags are stored in the high-order byte and lock count in the low-order byte of the return value. Use the GMEM_LOCKCOUNT mask on the return value to retrieve the lock count. If the handle hmem, does not exist or is invalid, GlobalFlags() and LocalFlags() return zero.

326.4 Errors
None.

326.5 Cross-References
None.

327 GlobalHandle, LocalHandle

327.1 Synopsis
DWORD GlobalHandle(LPVOID lpaddress);
DWORD LocalHandle(LPVOID lpaddress);

327.2 Description
The GlobalHandle() function searches for a memory segment that contains lpaddress, and returns its associated handle. GlobalHandle() is functionally equivalent to GlobalHandle32().

The LocalHandle() function simply calls GlobalHandle().
327.3 Returns

GlobalHandle() and LocalHandle() return the handle associated with the memory segment that contains lpaddress. (Note: lpaddress can be on the boundary or in the middle of the memory segment.) GlobalHandle() and LocalHandle() return zero, if they are unsuccessful.

327.4 Errors

None.

327.5 Cross-References

None.

328 GlobalLock, GlobalUnlock, LocalLock, LocalUnlock

328.1 Synopsis

LPVOID GlobalLock(HANDLE hMem);
BOOL GlobalUnlock(HANDLE hMem);
LPVOID LocalLock(HANDLE hMem);
BOOL LocalUnlock(HANDLE hMem);

328.2 Description

The GlobalLock() and LocalLock() functions increment the lock count and lock the memory specified by the hMem parameter. Locked memory cannot be moved or discarded unless reallocated by GlobalReAlloc() or LocalReAlloc(). The memory segment remains locked until its lock count decreases to zero.

The GlobalUnlock() and LocalUnlock() functions decrement the lock count and unlock the memory specified by the hMem parameter. See GlobalFlags() or LocalFlags() to find the number of locks on a memory segment.

328.3 Returns

GlobalLock() and LocalLock() return a pointer to the memory segment, if they are successful.

GlobalUnlock() and LocalUnlock() return FALSE if the lock count for hMem reaches zero. Otherwise, GlobalUnlock() and LocalUnlock() return a TRUE value.

328.4 Errors

None.

328.5 Cross-References

None.

329 GlobalLRUNewest, GlobalLRUOldest

329.1 Synopsis

HGLOBAL GlobalLRUNewest(HGLOBAL hglb);
HGLOBAL GlobalLRUOldest(HGLOBAL hglb);

329.2 Description

The GlobalLRUNewest() function moves a memory segment to the newest LRU (least-recently-used) position in memory. This reduces the likelihood of it being discarded soon. GlobalLRUOldest() moves a memory segment to the oldest LRU position in memory. This increases the likelihood of it being discarded soon. GlobalLRUNewest() and GlobalLRUOldest() are unnecessary in a virtual memory environment. Therefore, they currently do nothing.

329.3 Returns

GlobalLRUNewest() and GlobalLRUOldest() both return value of hglb.

329.4 Errors

None.
329.5 Cross-References

None.

330 GlobalNotify, NotifyProc

330.1 Synopsis

void GlobalNotify(GNOTIFYPROC NotifyProc);
BOOL CALLBACK NotifyProc(HGLOBAL hMem);

330.2 Description

The GlobalNotify() function sets the callback function pointed to by the NotifyProc parameter. If GlobalNotify() is called more than once, only the last installed procedure is notified.

The NotifyProc() function is a user-defined, exported callback function of type GNOTIFYPROC, which is called by the system whenever a global memory segment, allocated with the GMEM_NOTIFY flag, is about to be discarded. The function is passed to the memory segment's handle in its hMem parameter. NotifyProc() should not assume its using the same stack segment as the application, nor should it call any routine that might move in memory. NotifyProc() must be in a fixed code segment in a DDL.

The system does not call the notification procedure when discarding memory belonging to a DLL.

If the memory segment is discarded, the application should use the GMEM_NOTIFY flag when calling GlobalReAlloc() so that it will be notified when the object is discarded again.

330.3 Returns

NotifyProc() returns TRUE, if the memory segment should be discarded. If the function returns FALSE, the block will not be discarded. GlobalNotify() does not return a value.

330.4 Errors

None.

330.5 Cross-References

None.

331 GlobalReAlloc

331.1 Synopsis

HGLOBAL GlobalReAlloc(HGLOBAL hMem, DWORD dwBytes, UINT uFlags);

331.2 Description

The GlobalReAlloc() function modifies the size or other attributes in a memory segment, specified by the hMem parameter. If GMEM_MODIFY is not flagged in the uFlags parameter, GlobalReAlloc() resizes the memory segment to dwBytes. The uFlags parameter can be a combination of the following values:

<table>
<thead>
<tr>
<th>uFlags values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMEM_DISCARDABLE(^1)</td>
<td>This flag makes a previously moveable memory segment discardable. (Can only be used with GMEM_MODIFY.)</td>
</tr>
<tr>
<td>GMEM_MODIFY</td>
<td>This flag allows modification of the memory segment's flags only (dwBytes is ignored); it can be used with GMEM_DISCARDABLE and GMEM_MOVEABLE.</td>
</tr>
<tr>
<td>GMEM_MOVEABLE(^1)</td>
<td>If a moveable memory segment is locked, this flag allows the segment to be moved to a new locked location without invalidating the handle. When it is used with GMEM_MODIFY, GlobalReAlloc() changes fixed memory to moveable memory.</td>
</tr>
</tbody>
</table>

\(^1\) Note: GMEM_DISCARDABLE and GMEM_MOVEABLE flags are mutually exclusive. Using both in the same call will result in an error.
If the `dwBytes` parameter is non-zero and the segment specified by `hMem` is fixed, `GlobalReAlloc()` will relocate the memory segment to a new fixed location.

A previously moveable and discardable segment will be discarded if `dwBytes` and the memory segment's lock count are both zero.

A previously moveable and discardable segment will be discarded if `dwBytes` and the memory segment's lock count are both zero.

`GlobalReAlloc()` fails if `dwBytes` and the memory segment is not moveable and discardable.

### Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMEM_NODISCARD</td>
<td>This flag prevents memory from being discarded if there is insufficient memory available. (Cannot be used with GMEM_MODIFY.)</td>
</tr>
<tr>
<td>GMEM_ZEROINIT</td>
<td>This flag initializes additional memory to zero, if memory is being added to the segment. GMEM_ZEROINIT cannot be used with GMEM_MODIFY.</td>
</tr>
</tbody>
</table>

1 - These options are not necessary in a VM environment. In most implementations, they will simply be ignored.

### 331.3 Returns

`GlobalReAlloc()` returns the handle of the reallocated memory segment, if it is successful. If unsuccessful, `GlobalReAlloc()` returns zero.

### 331.4 Errors

None.

### 331.5 Cross-References

`GlobalAlloc()`

### 332 GlobalSize, LocalSize

#### 332.1 Synopsis

`DWORD GlobalSize(HANDLE hMem);`

`DWORD LocalSize(HANDLE hMem);`

#### 332.2 Description

The `GlobalSize()` and `LocalSize()` functions return the size of the memory segment specified by the `hMem` parameter.

`LocalSize()` calls `GlobalSize()`.

#### 332.3 Returns

The `GlobalSize()` and `LocalSize()` functions return the size (in bytes) of the memory segment specified by `hMem`, if they are successful. If the handle `hMem` is not valid or the memory segment has been discarded, the `GlobalSize()` and `LocalSize()` functions return zero.

#### 332.4 Errors

None.

#### 332.5 Cross-References

None.

### 333 LocalInit, LocalShrink

#### 333.1 Synopsis

`BOOL LocalInit(UINT)`
BOOL LocalShrink(UINT)

333.2 Description
The LocalInit() and LocalShrink() functions are provided as stub functions that perform no actions.

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

333.4 Errors
None.

333.5 Cross-References
None.

334 Catch, Throw

334.1 Synopsis
int Catch(LPINT lpCatchBuf);
void Throw(LPINT lpCatchBuf, int nReturnCode);

334.2 Description
The Catch() function saves the current execution environment and stores it in the buffer. The lpCatchBuf parameter points to the buffer. Catch() is similar to the C library function setjmp.

Throw() restores the execution environment from the buffer specified by the lpCatchBuf parameter. Throw() is similar to the C library function longjmp.

The execution environment is composed of the contents of all system registers and the instruction pointer. The execution environment is copied into the CATCHBUF structure in the lpCatchBuf buffer.

334.3 Returns
Catch() returns zero after being called. When the Throw() function is called, the environment is restored. Execution resumes from the point where the Catch() function returns again. This time, the return value of the Catch() function is the nReturnCode value, passed to Throw() as a parameter.

Throw() never returns, except to send requested values to Catch().

334.4 Errors
None.

334.5 Cross-References
None.

335 Yield, DirectedYield

335.1 Synopsis
void Yield(void);
void DirectedYield(hTask);

335.2 Description
The Yield() and DirectedYield() functions are used to pass control between multiple running tasks.

Yield() suspends the execution of the current task and passes the control to a waiting task that has messages waiting in the message queue.

DirectedYield() passes control to a task specified in the hTask parameter. The task is activated only if there is an event in its queue. If no events are queued for the specified task, the control is passed to another waiting task. To force the task to be activated regardless of the event status, the PostAppMessage() function should be called with WM_NULL as a message identifier before calling DirectedYield(), so that an event is placed into the task's queue.
These functions return when the task regains control.

335.3 Returns
None.

335.4 Errors
None.

335.5 Cross-References
PostAppMessage()

336 GetCurrentTask
336.1 Synopsis
HTASK GetCurrentTask(void);

336.2 Description
The GetCurrentTask() function retrieves the handle of the currently running task.

336.3 Returns
This function returns the handle of the current task.

336.4 Errors
None.

336.5 Cross-References
GetWindowTask()

337 GetNumTasks
337.1 Synopsis
UINT GetNumTasks(void);

337.2 Description
The GetNumTasks() function returns the number of tasks currently running in the system.

337.3 Returns
This function returns the number of running tasks.

337.4 Errors
None.

337.5 Cross-References
None.

338 GetWindowTask
338.1 Synopsis
HTASK GetWindowTask(HWND hWnd);

338.2 Description
The GetWindowTask() function retrieves the handle of a task that created the window specified in hWnd parameter.

338.3 Returns
This function returns the task handle, if it is successful. Otherwise, it returns zero.
338.4 Errors
None.

338.5 Cross-References
EnumTaskWindows(), GetCurrentTask()

339 IsTask
339.1 Synopsis
BOOL IsTask(HTASK hTask);

339.2 Description
The IsTask() function checks whether the task handle specified in hTask parameter is valid.

339.3 Returns
This function returns TRUE, if the task handle is valid. Otherwise, it returns FALSE.

339.4 Errors
None.

339.5 Cross-References
None.

340 WinHelp
340.1 Synopsis
BOOL WinHelp(HWND hwnd, LPCSTR lpszHelpFile, UINT fuCommand, DWORD dwData);

340.2 Description
The WinHelp() function invokes the Windows Help facility and optionally requests the application specific help topic. The lpszHelpFile parameter specifies the name of the help file that the application is about to display. The fuCommand parameter can be as follows:

<table>
<thead>
<tr>
<th>HELP_CONTEXT</th>
<th>This value displays the help for a particular topic; the dwData parameter should contain the context number for the topic requested.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELP_CONTENTS</td>
<td>This value displays the help contents; the dwData parameter is ignored.</td>
</tr>
<tr>
<td>HELP_SETCONTENTS</td>
<td>This value determines the contents topic that should be displayed when a user presses F1 key; the dwData parameter should contain the context number for the topic requested as the contents topic.</td>
</tr>
<tr>
<td>HELP_SETCONTEXTPOPUP</td>
<td>This value displays a pop-up window with the particular help topic; the dwData parameter should contain the context number for the topic requested.</td>
</tr>
<tr>
<td>HELP_KEY</td>
<td>This value displays the topic that matches one found in the help's keyword list. The dwData parameter should point to a string with the target keyword; if more than one keyword is found, the help displays the Search dialog with the topics listed in the GoTo list box.</td>
</tr>
<tr>
<td>HELP_PARTIALKEY</td>
<td>This value displays the topic found in the help's keyword list. If the dwData parameter points to a string with the target keyword and more than one keyword is found, the help displays the Search dialog with the topics listed in the GoTo list box. If the dwData parameter points to an empty string, the help brings up the empty Search dialog with no keywords in it.</td>
</tr>
</tbody>
</table>
HELP_MULTIKEY This value displays the topic found in the alternate help's keyword list; the dwData parameter should point to MULTIKEYHELP structure, which specifies the footnote character and the keyword.

HELP_COMMAND This value executes the help macro; the dwData parameter should point to the character string with the macro to be executed.

HELP_SETWINPOS This value displays and positions the help window according to the data passed in dwData parameter; the dwData parameter should point to HELPWININFO structure, which specifies the size and position of the primary or secondary help windows.

HELP_FORCEFILE This value tries to open the correct help file; if the file is already open by Help, there is no action and the dwData parameter is ignored.

HELP_QUIT If no other applications have requested help, the Help application is closed and the dwData parameter is ignored.

340.3 Returns
The WinHelp() function returns TRUE if it is successful. Otherwise, it returns FALSE.

340.4 Errors
None.

340.5 Cross-References
None.

341 EnumTaskWindows, EnumTaskWndProc

341.1 Synopsis
BOOL EnumTaskWindows(HTASK hTask, WNDENUMPROC EnumTaskWndProc, LPARAM lParam);
BOOL CALLBACK EnumTaskWndProc(HWND hWnd, LPARAM lParam);

341.2 Description
The EnumTaskWindows() function enumerates all windows associated with a task. The hTask parameter contains the handle to the task. The lParam parameter contains a user-defined value. The EnumTaskWndProc parameter is a pointer to an exported, user-defined, callback function that is called each time the EnumTaskWindows() function finds a window associated with the task. The EnumTaskWindows() function passes the window's handle and the value of the lParam parameter to the callback function. The process continues until all of the task's windows are enumerated or until the EnumTaskWndProc callback function returns FALSE. The EnumTaskWindows() function enumerates all top-level windows and does not consider child windows during its search.

The EnumTaskWndProc() function is an exported, user-defined, callback function of type WNDENUMPROC whose address is passed to the EnumTaskWindows() function. The EnumTaskWindows() function calls the EnumTaskWndProc() function each time that it finds a window associated with the task. The hWnd parameter is a handle to a window associated with a task. The lParam parameter is a user-defined value that was passed to the EnumTaskWindows() function when it was called.

341.3 Returns
If the EnumTaskWindows() function is successful it returns TRUE. If the EnumTaskWindows() function is not successful, it returns FALSE.

The EnumTaskWndProc() function should return TRUE to inform the EnumTaskWindows() function to continue enumerating the task's windows. The EnumTaskWndProc() function should return FALSE to inform the EnumTaskWindows() function to stop enumerating the task's windows.

341.4 Errors
None.
342 WinExec

342.1 Synopsis
UINT WinExec(LPCSTR lpszCmdLine, UINT uiCmdShow);

342.2 Description
The WinExec() function starts an application. The functionality is similar to LoadModule().

The lpszCmdLine parameter is a pointer to the command line string, providing the name of the executable file and optional command-line parameters.

The uiCmdShow parameter specifies the show style for the new application. It is similar to the argument used in the ShowWindow() function.

If the lpszCmdLine string does not contain the full path, the following directories are searched:
- the current directory
- the Windows directory (retrieved by GetWindowsDirectory())
- the system directory (retrieved by GetSystemDirectory())
- the directory containing the executable file for the current task (retrieved by GetModuleFileName())
- the directories listed in the PATH environment variable
- the directories mapped in the network

342.3 Returns
This function returns an instance handle of the loaded module, if successful. Otherwise, an error value less than HINSTANCE_ERROR is returned.

342.4 Errors
The cause of failure of the WinExec() function and the error codes can be as follows:

0 There is insufficient memory to load the module or corrupted executable file.
2 The file is not found.
4 The path is not found.
5 A sharing or network-protection error has occurred.
8 There is insufficient memory to start the application.
11 An invalid executable file was discovered.
14 An unknown type of executable file was discovered.
16 A second attempt was made to load an executable file with multiple data segments not marked read-only.
19 The file is compressed.
20 A DLL file is invalid or one of the DLLs it requires is corrupt.
21 The library module needs 32-bit extensions not provided by the system.

342.5 Cross-References
LoadModule(), FreeModule()

343 WinMain

343.1 Synopsis
int WinMain(HINSTANCE hInst, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
343.2 Description
The WinMain() function is an initial entry point of an application and is called by the system to start the program. The hInst parameter specifies the data instance of the application. The hPrevInst parameter is a previous instance of the application, if the application is already running. The lpszCmdLine parameter points to the command-line string for the application. The nCmdShow parameter determines how the application's main window will be shown. The options are the same as they are for the nCmdShow parameter of the ShowWindow() function.

WinMain() of the application performs all necessary instance-specific initialization. If the first instance of the application is being started (the hPrevInstance parameter is 0), it might also need to do some application initialization. After initialization it provides the message loop that drives the application's execution. This loop is responsible for dispatching the messages and yielding control to other tasks. The normal termination of the task happens when it receives a WM_QUIT message. In response, the WinMain() function exits with the return value passed by the PostQuitMessage() function.

343.3 Returns
WinMain() returns the value passed to the PostQuitMessage() function or zero if it returns before entering the message loop.

343.4 Errors
None.

343.5 Cross-References
GetMessage(), PostQuitMessage(), ShowWindow()

344 ExitWindows
344.1 Synopsis
BOOL ExitWindows(DWORD dwRetCode, UINT reserved);

344.2 Description
The ExitWindows() function shuts down the runtime environment with an option to restart it. The dwRetCode parameter specifies the way the system should be shut down. The high-order word of dwRetCode should be zero. The low-order word is the value to be returned by the system on exit. If the low-order code is EW_RESTARTWINDOWS, the system runtime should be restarted. If it is EW_REBOOTSYSTEM, the requested action is to restart the computer, which is implementation-dependent.

The reserved parameter is not used and should be set to zero.

In response to the call to ExitWindows(), the system sends the WM_QUERYENDSESSION message to all running tasks. If one or more tasks return 0, the system is not shut down. If all tasks return non-zero, the system sends the WM_ENDSESSION message to all tasks and terminates.

344.3 Returns
The ExitWindows() function returns FALSE if one or more tasks will not terminate. If the system is being shut down, the function does not return.

344.4 Errors
None.

344.5 Cross-References
None.

345 GetAsyncKeyState
345.1 Synopsis
int GetAsyncKeyState(int keycode);
345.2 Description
The *GetAsyncKeyState()* function indicates, at the time the function is called, whether a particular key is up or down. It also indicates if the key was pressed since the last call to the *GetAsyncKeyState()* function. The keycode parameter can have one of the 256 possible virtual-key codes. In the case where the keycode parameter contains the value VK_LBUTTON or VK_RBUTTON, the status of the left or right mouse button is returned respectively, regardless of whether the *SwapMouseButton()* function had been called to redefine the meaning of the left and right mouse buttons.

345.3 Returns
The function return value indicates if the key was pressed since the last call to *GetAsyncKeyState()* and the status of the key (UP or DOWN). If the most significant bit is set then the key is down. If the least significant bit is set then that particular key has been pressed since the last call to the function *GetAsyncKeyState()*.

345.4 Errors
None.

345.5 Cross-References
*GetKeyboardState(), GetKeyState(), SetKeyboardState(), SwapMouseButton()*

346 qGetInputState
346.1 Synopsis
BOOL qGetInputState(void);

346.2 Description
The *qGetInputState()* function checks the system queue and identifies mouse clicks or keyboard events that need to be processed. It should be mentioned here that the system stores keyboard events (which occur when one or more keys are pressed) and mouse events in the system queue and determines whether there are mouse clicks or keyboard events in the system.

346.3 Returns
The function returns TRUE if mouse clicks or keyboard events are found in the system queue. Otherwise, it returns FALSE.

346.4 Errors
None.

346.5 Cross-References
*EnableHardwareInput()*

347 GetKeyboardState, SetKeyboardState
347.1 Synopsis
void GetKeyboardState(BYTE *lpKeyStateBuf);
void SetKeyboardState(BYTE *lpKeyStateBuf);

347.2 Description
The *GetKeyboardState()* function copies the status of the 256 virtual-keyboard keys to the buffer. The *lpKeyStateBuf* parameter contains a pointer to the 256-byte buffer into which the function copies the virtual-key codes. *GetKeyboardState()* is called when a keyboard-input message is generated and retrieves the state of the keyboard at the time the message is generated. A key is considered to be down if the high-order bit is set to 1. Otherwise, the key is considered to be up. If the low-order bit is set to 1, the key is considered to be toggled. In the case of toggle keys like NUMLOCK, SCROLL LOCK and CAPSLOCK, it is considered toggled if the key has been pressed an odd number of times since the system was started, and is considered untoggled if the low-order bit is set to zero.
The `SetKeyboardState()` function copies the contents of the 256-byte array buffer pointed to by the `lpKeyStateBuf` parameter into the system keyboard state table. The `lpKeyStateBuf` parameter contains a pointer to the 256-byte array that contains the keyboard key states. Typically an application calls the function `GetKeyboardState()` to obtain the keyboard key states and then modifies the desired bytes before calling the `SetKeyboardState()` function. In the case of the NUMLOCK, CAPSLOCK, and SCROLL LOCK keys, the BIOS flags and LED's are set according to the values of the VK_NUMLOCK, VK_CAPITAL, and VK_SCROLL entries of the array, respectively.

### 347.3 Returns

- `GetKeyboardState()` does not return a value.
- `SetKeyboardState()` does not return a value.

### 347.4 Errors

None.

### 347.5 Cross-References

- `GetKeyState()`

---

### 348 GetKeyNameText

#### 348.1 Synopsis

```c
int GetKeyNameText(LONG lParam, LPSTR lpszKeyBuffer, int nbMaxKey);
```

#### 348.2 Description

The `GetKeyNameText()` function retrieves a string representing the name of the key. The `lParam` parameter identifies the 32-bit parameter of the keyboard message that needs to be processed. The `lpszKeyBuffer` parameter contains a pointer to the buffer in which the key name will be stored. The `nbMaxKey` parameter is the maximum length in bytes of the keyname less the null-terminating character. The value of this parameter is usually the size of the buffer identified by the `lpszKeyBuffer` parameter minus one. The current keyboard driver that is in use determines the format of the key-name string. If the name of the key is longer than one character, the driver maintains a list in the form of character strings. The key name is translated into the principal language supported by the keyboard driver depending on the layout of the currently installed keyboard device.

#### 348.3 Returns

This function returns the length of the string in bytes that was copied to the buffer identified by the `lpszKeyBuffer` parameter, if it is successful. Otherwise, it returns zero.

#### 348.4 Errors

None.

#### 348.5 Cross-References

None.

---

### 349 GetKeyState

#### 349.1 Synopsis

```c
int GetKeyState(int vidkey);
```

#### 349.2 Description

The `GetKeyState()` function obtains the state of the virtual key identified by the `vidkey` parameter. The obtained state identifies if the key state is up, down, or toggled. The `vidkey` parameter identifies the virtual key. This parameter should be set to the ASCII value of the character if the virtual key is a letter or digit (A-Z or 1-9), otherwise it must be set to the virtual-key code.

This function is called when the keyboard-input message is sent and obtains the state of the key at the time the input message is generated.
349.3 Returns
The function returns a value which identifies the state of the given virtual key. Depending on the value of the high and low-order bits, the key may be up or down or toggled. The key is considered to be down if the high-order bit is 1. Otherwise, it is considered to be up. A key is considered to be in a toggled state if the low-order bit is 1. Otherwise, it is considered to be untoggled. A key is considered to be toggled if it has been pressed an odd number of times since the system was started.

349.4 Errors
None.

349.5 Cross-References
GetAsyncKeyState(), GetKeyboardState()

350 GetKBCodePage
350.1 Synopsis
int GetKBCodePage(void);

350.2 Description
The GetKBCodePage() function returns the current system code page. This function is actually provided by the keyboard driver. Therefore, an application that intends to call this function should include the following two lines in the module definition file (.DEF):

IMPORTS
KEYBOARD.GETKBPCODEPAGE

The file OEMANSI.BIN, if present, resides in the system directory and is read by the system before it overwrites the OEM/ANSI translation tables in the keyboard driver. If the language that is chosen by the Setup program does not use the default code page then the corresponding file for that language is copied into the OEMANSI.BIN file in the system directory. If the selected language uses the default code page then the file OEMANSI.BIN is deleted from the system directory if one is present.

350.3 Returns
If the function is successful then it returns the code page that is currently in use by the system. The return value will identify which code page is being used as listed below.

437 Default (United States, used by most countries: indicates that there is no OEMANSI.BIN in the Windows directory)
850 International (OEMANSI.BIN = XLAT850.BIN)
860 Portugal (OEMANSI.BIN = XLAT860.BIN)
861 Iceland (OEMANSI.BIN = XLAT861.BIN)
863 French Canadian (OEMANSI.BIN = XLAT863.BIN)
865 Norway/Denmark (OEMANSI.BIN = XLAT865.BIN)

350.4 Errors
None.

350.5 Cross-References
GetKeyboardType()

351 OemKeyScan
351.1 Synopsis
DWORD OemKeyScan(UINT idOemChar);
351.2 Description
The `OemKeyScan()` function converts OEM ASCII codes 0 through 0xFF to their corresponding OEM scan codes and shift states. The `idOemChar` parameter identifies the ASCII value of the OEM character. Characters that require CTRL+ALT or dead keys are not translated by this function, but must instead be copied by simulating the input using the ALT+keypad mechanism, with the NUM LOCK key in the off position. In later versions of the keyboard device drivers, this function calls the `VkKeyScan()` function. The `OemKeyScan()` function is used primarily to send OEM text to another application by simulating keyboard input.

351.3 Returns
The interpretation of the low-order and high-order word identifies the information returned. The low-order word of the return value identifies the scan code of the specified OEM character. The high-order word contains the flags identifying the shift state:
- the SHIFT key has been pressed if bit 1 has been set
- the CTRL key has been pressed if bit 2 is set

If the character has not been defined in the OEM character tables then both the high- and low-order words will contain a value of -1.

351.4 Errors
None.

351.5 Cross-References
`VkKeyScan()`

352 MapVirtualKey
352.1 Synopsis
`UINT MapVirtualKey(UINT idKeyCode, UINT KeyMapType);`

352.2 Description
The `MapVirtualKey()` function converts the virtual-key code identified by the `idKeyCode` parameter to the scan code or ASCII value, or vice-versa. The `idKeyCode` parameter identifies the virtual-key code or scan code for the key. The interpretation of this parameter depends on the value of the `KeyMapType` parameter. The `KeyMapType` parameter identifies the translation that has to be performed. If the value of this parameter is 1, the `idKeyCode` is identified as a scan code and is translated into a virtual-key code. If the value of the parameter is 2, the `idKeyCode` is identified as a virtual-key code and is translated to an unshifted ASCII value. Any other value that this parameter can have is reserved.

352.3 Returns
The return value of this function depends on the value of the parameters `idKeyCode` and `KeyMapType`.

352.4 Errors
None.

352.5 Cross-References
`OemKeyScan()`, `VkKeyScan()`

353 VkKeyScan
353.1 Synopsis
`UINT VkKeyScan(UINT idChar);`

353.2 Description
The `VkKeyScan()` function converts a system character to a virtual-key code and shift state for the keyboard. The `idChar` parameter identifies the character that needs to be converted. This function is most often used to force conversion for the main keyboard only. Therefore, the numeric keypad values (VK_NUMPAD0 through
VK_DIVIDE) are ignored and not converted by this function. This function is also used by an application to send characters, by using the WM_KEYUP and WM_KEYDOWN messages.

353.3 Returns
The virtual-key code is returned in the low-order byte and the shift state is returned in the high-order byte, if this function is successful. The shift state can have on the following values:

1. The character is shifted.
2. The character is a control character.
3-5. A Shift-key combination that is not used for characters.
6. The character is generated by the CTRL+ALT key combination.
7. The character is generated by the SHIFT+CTRL+ALT key combination.

If no key is found that can be translated to the virtual key, the function returns -1.

353.4 Errors
None.

353.5 Cross-References
OemKeyScan()
IMPORTS
KEYBOARD.GETKEYBOARDTYPE

The KbTypeInfo parameter can have one of the following values:

0  This value fetches the type of keyboard.
1  This value fetches the subtype of the keyboard.
2  This value fetches the total number of function keys on the keyboard.

When the subtype is fetched by this function, it can be one of the following values.

Note: The subtype value is OEM-dependent.

1  IBM PC/XT, or compatible (83-key) keyboard
2  Olivetti "ICO" (102-key) keyboard
3  IBM AT (84-key) or similar keyboard
4  IBM Enhanced (101- or 102-key) keyboard
5  Nokia 1050 and similar keyboards
6  Nokia 9140 and similar keyboards
7  Japanese keyboard

When the number of function keys is fetched by this function, it can be one of the following values, for each of the seven keyboard types.

1  10
2  12 (sometimes 18)
3  10
4  12
5  10
6  24
7  This is a hardware-dependent value and must be specified by the OEM.

355.3  Returns
If the function GetKeyboardType() is successful it returns the requested information. Otherwise, it returns zero.

355.4  Errors
None.

355.5  Cross-References
None.

356  FindResource

356.1  Synopsis
HRSRC FindResource(HINSTANCE hInstance, LPCSTR Name, LPCSTR Type);

356.2  Description
The FindResource() function returns a handle to a resource in a module. The hInstance parameter is the instance of the module whose that contains the resource. The parameter name is the pointer to a null-terminated string containing the name of the desired resource. The type parameter is the resource type of the desired resource. The type parameter can be one of the following system defined values:

RT_ACCELERATOR  accelerator table resource
RT_BITMAP       bitmap resource
RT_CURSOR        cursor resource
RT_DIALOG        dialog box resource
To reduce the amount of memory required for the resources used by an application, the application can refer to a resource by its integer identifier instead of by its name. You can pass an identifier to the `FindResource()` function in two different ways.

If the name or type parameter's high-order word is zero, the integer identifier of the name or type of the resource can be specified in the parameter's low-order word. If the name or type parameter's high-order word is not zero, it is assumed to be a point to a string.

If the first character of the string is a pound sign (#), the other characters in the string can form a decimal number that is the integer identifier of the resource's name or type. The string #343, for example, is the resource identifier, 343.

356.3 Returns
If the `FindResource()` function is successful, it returns a handle to the resource. If the `FindResource()` function is not successful, it returns NULL.

356.4 Errors
None.

356.5 Cross-References
None.

357 LoadResource, FreeResource

357.1 Synopsis
HGLOBAL LoadResource(HINSTANCE hInstance, HRSRC hResource);
BOOL FreeResource(HGLOBAL hGlobal);

357.2 Description
The `LoadResource()` function loads a resource into global memory and returns a handle to the memory. The hInstance parameter is the instance of the module that contains the resource. The hResource parameter is a handle of the desired resource retrieved by using the `FindResource()` function.

The `FreeResource()` function frees a resource previously loaded by the `LoadResource()` function. The hGlobal parameter is assumed to be the memory handle returned by the `LoadResource()` function when the resource was loaded.

357.3 Returns
If the `LoadResource()` function is successful, it returns a handle to the global memory containing the resource's data. If the `LoadResource()` function is not successful, it returns NULL. If the `FreeResource()` function is successful, it returns TRUE. Otherwise, it returns FALSE.

357.4 Errors
None.

357.5 Cross-References
`FindResource()`
358  LockResource
358.1  Synopsis
  void *LockResource(HGLOBAL hGlobal);

358.2  Description
  The LockResource() function locks a resource that has been loaded into global memory and returns a pointer to the
  resource's data. The hGlobal parameter is assumed to be the memory handle returned by the LoadResource() function when the resource was loaded. A resource's memory will not be discarded while it is locked.

358.3  Returns
  If the LockResource() function is successful, it returns a pointer to the resource's data. If the LockResource() function is not successful, it returns NULL.

358.4  Errors
  None.

358.5  Cross-References
  LoadResource()

359  LoadString
359.1  Synopsis
  int LoadString(HINSTANCE hInstance, UINT ResourceID, LPSTR Buffer, int Count);

359.2  Description
  The LoadString() function loads a string resource into a given buffer. The hInstance parameter is the instance of the
  module that contains the resource. The ResourceID parameter contains the identifier associated with the resource. The Buffer parameter is a pointer to a memory buffer where the string is copied. The Count parameter contains the size of the buffer.

359.3  Returns
  If the LoadString() function is successful, it returns the number of bytes in the string that were copied into the buffer. If the LoadString() function is not successful, it returns zero.

359.4  Errors
  None.

359.5  Cross-References
  LoadResource()

360  LoadIcon
360.1  Synopsis
  HICON LoadIcon(HINSTANCE hInstance, LPCSTR ResourceID);

360.2  Description
  The LoadIcon() function loads an icon resource from a module or one of the predefined system icons.
  The hInstance parameter is the instance of the module that contains the resource. If a system icon is being loaded, the value of the hInstance parameter should be NULL.
  The ResourceID parameter specifies which icon resource to load and can be used in one of two different ways. If you want to refer to the resource by name, the parameter can be a pointer to a null-terminated string containing the name of the icon resource. If you want to refer to the resource using an identifier, you can specify the icon resource's identifier in the parameter's low-word and its high-word should be set to zero. The MAKEINTRESOURCE macro can be used to create this value.
The following *ResourceId* values can be used to load a system icon:

- IDI_APPLICATION: generic application icon
- IDI_ASTERISK: asterisk icon
- IDI_EXCLAMATION: exclamation point icon
- IDI_HAND: hand-shaped icon
- IDI_QUESTION: question mark icon

An application should destroy any icons that it loads, by calling the *DestroyIcon()* function. System icons, however, do not have to be destroyed.

### 360.3 Returns

If the *LoadIcon()* function is successful, it returns a handle to the loaded icon. Otherwise, it returns NULL.

### 360.4 Errors

None.

### 360.5 Cross-References

- *DestroyIcon()*

---

## 361 LoadBitmap

### 361.1 Synopsis

```c
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCSTR ResourceID);
```

### 361.2 Description

The *LoadBitmap()* function loads a bitmap resource from a module or one of the predefined system bitmaps.

The *hInstance* parameter is the instance of the module that contains the resource. If a system bitmap is being loaded, the value of the *hInstance* parameter should be NULL.

The *ResourceId* parameter specifies which bitmap resource to load from the file and can be used in one of two different ways. If you want to refer to the resource by name, the parameter can be a pointer to a null-terminated string containing the name of the bitmap resource. If you want to refer to the resource using an identifier, you can specify the bitmap resource's identifier in the parameter's low-word and its high-word should be set to zero. The MAKEINTRESOURCE macro can be used to create this value.

The following *ResourceId* values can be used to load a system bitmap:

- OBM_BTNCORNERS: OBM_OLD_RESTORE
- OBM_BTSIZE: OBM_OLD_RGARROW
- OBM_CHECK: OBM_OLD_UPARROW
- OBM_CHECKBOXES: OBM_OLD_ZOOM
- OBM_CLOSE: OBM_REDUCE
- OBM_COMBO: OBM_REDUCED
- OBM_DNARROW: OBM_RESTORE
- OBM_DNARROWD: OBM_RESTORED
- OBM_DNARROWI: OBM_RGARROW
- OBM_LFARROW: OBM_RGARROWD
- OBM_LFARROWD: OBM_RGARROWI
- OBM_LFARROWI: OBM_SIZE
- OBM_MNARROW: OBM_UPARROW
In order to use one of the system bitmap values listed above, the constant OEMRESOURCE must be defined before
the header file WINDOWS.H is included.

An application should destroy all bitmaps, even system bitmaps, that it loads by calling the `DeleteObject()` function.

### 361.3 Returns

If `LoadBitmap()` is successful, it returns a handle to the loaded bitmap. Otherwise, it returns NULL.

### 361.4 Errors

None.

### 361.5 Cross-References

`DeleteObject()`

---

### 362 SetResourceHandler, LoadProc

#### 362.1 Synopsis

```c
RSRCHDLRPROC SetResourceHandler(HINSTANCE hInstance, LPCSTR ResourceType,
                                RSRCHDLRPROC LoadProc);
HGLOCAL CALLBACK LoadProc(HGLOBAL hResMem, HINSTANCE hInstance, HRSRC hResource);
```

#### 362.2 Description

The `SetResourceHandler()` function can be used to install a callback function that loads resources. The `hInstance` parameter is the instance of the module whose file contains the resource. The `ResourceType` parameter specifies the type of resource. For predefined resource types, the parameter's low-word should contain the resource type and its high-word should be set to zero. The MAKEINTRESOURCE macro can be used to create this value. The `LoadProc` parameter contains a procedure-instance address of a `RSRCHDLRPROC` callback function whose name has been exported in the application's module-definition file.

A user-defined `LoadProc()` callback function receives information about a resource to be locked. The `hResMem` parameter is a handle to memory containing the resource. If the value of `hResMem` is NULL, the resource is not loaded into memory. If an error occurs when locking `hResMem`, the resource has been discarded and needs to be reloaded into memory. The `hInstance` parameter is the instance of the module whose executable file contains the resource. The `hResource` parameter is a handle of the resource created by using the `FindResource()` function.

#### 362.3 Returns

If the `SetResourceHandler()` function is successful, it returns a handle to the previously installed resource handler. If no handler has been installed, the `SetResourceHandler()` returns a pointer to the system's default resource handler.

`SetResourceHandler()`'s return value is a global memory handle for memory that was allocated using the `GlobalAlloc()` function's GMEM_DDESHARE flag.

#### 362.4 Errors

None.

### 362.5 Cross-References

`FindResource()`
363 SizeofResource

363.1 Synopsis

DWORD SizeofResource(HINSTANCE hInstance, HRSRC hResource);

363.2 Description

The `SizeofResource()` function determines the size of a resource in bytes. The `hInstance` parameter is the instance of the module that contains the resource. The `hResource` parameter is a handle of the resource created by using the `FindResource()` function.

363.3 Returns

If `SizeofResource()` is successful, it returns the size of the resource in bytes. The value may be adjusted to a larger value due to memory alignment. If the `SizeofResource()` function is not successful, it returns zero.

363.4 Errors

None.

363.5 Cross-References

`FindResource()`

364 LoadMenu

364.1 Synopsis

HMENU LoadMenu(HINSTANCE hInstance, LPCSTR ResourceID);

364.2 Description

The `LoadMenu()` function loads a menu resource from a module. The `hInstance` parameter is the instance of the module that contains the resource. The `ResourceId` parameter specifies which menu resource to load and can be used in one of two different ways. If you want to refer to the resource by name, the parameter can point to a null-terminated string containing the name of the menu resource. If you want to refer to the resource using an identifier, you can specify the menu resource's identifier in the parameter's low-word and its high-word should be set to zero. The `MAKEINTRESOURCE` macro can be used to create this value.

Before quitting, an application should use the function `DestroyMenu()` to free any menus that have not been associated with a window via the `SetMenu()` function.

364.3 Returns

If the `LoadMenu()` function is successful, it returns a handle to the loaded menu. If the `LoadMenu()` function is not successful, it returns NULL.

364.4 Errors

None.

364.5 Cross-References

`DestroyMenu()`, `SetMenu()`

365 LoadMenuIndirect

365.1 Synopsis

HMENU LoadMenuIndirect(const void *MenuInfo);

365.2 Description

The `LoadMenuIndirect()` function creates a menu resource from the information supplied to the function. The `MenuInfo` parameter is a pointer to a block of memory that contains information about the new menu resource. The memory block contains a `MENUITEMTEMPLATEHEADER` structure followed by one or more `MENUITEMTEMPLATE` structures.
Before quitting, an application should use the function `DestroyMenu()` to free any menus that are not associated with a window via the `SetMenu()` function.

365.3 **Returns**
If the `LoadMenuIndirect()` function is successful, it returns a handle to the loaded menu. If the `LoadMenuIndirect()` function is not successful, it returns NULL.

365.4 **Errors**
None.

365.5 **Cross-References**
None.

366 **LoadAccelerators**

366.1 **Synopsis**

```c
HACCEL LoadAccelerators(HINSTANCE hInstance, LPCSTR AccelTableName);
```

366.2 **Description**
The `LoadAccelerators()` function loads an accelerator table into memory and returns a handle to the accelerator table. The `hInstance` parameter is the instance of the module that contains the resource. The `AccelTableName` parameter is a pointer to a null-terminated string containing the name of the accelerator table.

366.3 **Returns**
If successful, the `LoadAccelerators()` function returns a handle to the accelerator table. If unsuccessful, the `LoadAccelerators()` function returns NULL.

366.4 **Errors**
None.

366.5 **Cross-References**

`LoadResource()`

367 **AllocResource**

367.1 **Synopsis**

```c
HGLOBAL AllocResource(HINSTANCE hinst, HRSRC hrsrc, DWORD cbResource);
```

367.2 **Description**
The `AllocResource()` function allocates uninitialized memory for the resource specified by the `hrsrc` parameter. The value of `hrsrc` should have been created by calling the `FindResource()` function.

367.3 **Returns**
`AllocResource()` returns the handle of the global memory block, if it is successful.

367.4 **Errors**
None.

367.5 **Cross-References**

`FindResource()`, `LoadResource()`, `AccessResource()`

368 **BuildCommDCB**

368.1 **Synopsis**

```c
int BuildCommDCB(LPCSTR lpszDevStr,DCB *lpDevcBlk);
```
368.2 Description
The BuildCommDCB() function converts the given device-definition control string into the appropriate serial device control block codes. The lpszDevcStr parameter contains a pointer to a character string which specifies the communication device control information. It should be noted that the format of the string should be the same as the format of the "mode" command that is used in MSDOS, to setup the Serial Communications port (COM 1 through COM 4).

The lpDevcBlk parameter contains a pointer to the Device Control Block structure (DCB).

368.3 Returns
If the BuildCommDCB() function is successful, it returns zero. Otherwise, it returns -1.

368.4 Errors
None.

368.5 Cross-References
SetComm()

369 ClearCommBreak, SetCommBreak
369.1 Synopsis
int ClearCommBreak (int NumComDev);
int SetCommBreak(int NumComDev);

369.2 Description
The ClearCommBreak() function returns the communications-device to nonbreak state and permits character transmission to take place. This function resets the break state of the communication device that has been set with the SetCommBreak() function. The NumComDev parameter specifies the communication device that is to be restored. This parameter is the return value of the function OpenComm().

The SetCommBreak() function puts the communication device specified by NumComDev in break state, suspending the transmission of characters. The NumComDev parameter, which is returned by the OpenComm() function, identifies the communication device that will be placed in the break state. The communication device can be released by the application from this break (suspended) state by calling the ClearCommBreak() function.

369.3 Returns
If ClearCommBreak() is successful, the return value is zero. It returns -1 if the NumComDev parameter contains an invalid device.

If the function SetCommBreak() is successful, the return value is zero. Otherwise, it is less than zero.

369.4 Errors
None.

369.5 Cross-References
OpenComm()

370 CloseComm, OpenComm
370.1 Synopsis
int CloseComm(int NumComDev);
int OpenComm(LPCSTR lpszDevcstr, UINT nbInQueue, UINT nbOutQueue);

370.2 Description
The CloseComm() function closes the communications device identified by NumComDev making sure that the output queue is empty. If any characters are still in the output queue, these are sent before the communication device is closed. Memory that was allocated for the device's transmission and receiving queues are freed. The NumComDev
parameter is the device identification value that the function OpenComm() returns and identifies the device that has to be closed.

The OpenComm() function is used to open the communication device. The lpszDevcstr parameter contains a pointer to the null-terminated string which identifies the communication device COMn (nth Serial communication port) or LPTn (nth Parallel communication port).

The nbInQueue parameter identifies the number of bytes the receiving queue can contain in the case of the COMn devices. This parameter is ignored if the device is a parallel port (LPTn).

The nbOutQueue parameter identifies the number of bytes the transmission queue can contain, in the case of COMn devices. This parameter is ignored if the device is a parallel port (LPTn).

In the system, serial communication ports COM ports 1 through 9 and Parallel ports 1 through 3 are also supported. OpenComm() fails if the device driver does not support the communication port number that needs to be used. When OpenComm() is called, the communication device is initialized to the default settings for the device, which change these settings and initialize the device to the new settings used by the SetCommState() function.

Since the parallel communication port devices are not interrupt driven, the parameters nbInQueue and nbOutQueue are ignored and the queue size is set to zero.

370.3 Returns
If CloseComm() is successful, the return value is set to zero. Otherwise, it returns a negative number.

If OpenComm() is successful, the return value is set to zero. Otherwise, it returns a negative number.

370.4 Errors
Other than the return value, no other error information is provided by the CloseComm() function.

The OpenComm() function can return any of the following errors:

IE_BADID The device identifier is invalid or unsupported.
IE_BAUDRATE The device baud rate is unsupported.
IE_BYTESSIZE The specified byte size is invalid.
IE_DEFAULT The default parameters are in error.
IE_HARDWARE The hardware is not available (is locked by another device).
IE_MEMORY The function cannot allocate the queues.
IE_NOPEN The device is not open.
IE_OPEN The device is already open.

If the OpenComm() function is called with both parameters nbInQueue and nbOutQueue set to zero, the return value is set to IE_MEMORY if the device is already open. It is set to IE_OPEN, if the device is already open.

370.5 Cross-References
SetCommState()
message that notifies the application that the input queue is ready to be read. If the value of this parameter is -1, the WM_COMMNOTIFY message is sent to the window identified by hwnd for CN_EVENT or CN_TRANSMIT notification only.

When a timeout occurs before the number of bytes identified in the nbWriteNotify parameter are sent, the CN_RECEIVE flag is set before the WM_COMMNOTIFY message is sent. If in a message, the CN_RECEIVE flag is set, the message is sent only when the size of the output queue is larger than the value of the nbOutQueue parameter. The nbOutQueue parameter specifies the minimum number of bytes that are there in the output queue. If the value of this parameter is -1, the WM_COMMNOTIFY message is sent to the window identified by hwnd only on a CN_EVENT or CN_RECEIVE notification. If the number of bytes in the output queue decreases to a value below the minimum value, the COM driver notifies the application by sending the notification message indicating that more information needs to be written to the output queue.

371.3 Returns
If the function is successful, TRUE is returned. A return value of FALSE indicates one of the following:
- the COM port identified by NumComDev is invalid
- an unopened port
- an unsupported function in COMM.DRV has been called

371.4 Errors
None.

371.5 Cross-References
WM_COMMNOTIFY

372 EscapeCommFunction

372.1 Synopsis
LONG EscapeCommFunction(int NumComDev, int NumFunction);

372.2 Description
The EscapeCommFunction() function requests the specified communication device to carry an extended function. OpenComm() returns the NumComDev parameter that identifies the device used to carry out the extended function. The NumFunction parameter identifies the function code of the extended function. The possible values are listed below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRDTR</td>
<td>This value clears the DTR (data-terminal-ready) signal.</td>
</tr>
<tr>
<td>CLRRTS</td>
<td>This value clears the RTS (request-to-send) signal.</td>
</tr>
<tr>
<td>GETMAXCOM</td>
<td>This value returns the maximum COM port identifier supported by the system; this value ranges from 0x00 to 0x7F, such that 0x00 corresponds to COM1, 0x01 to COM2, 0x02 to COM3, and so on.</td>
</tr>
<tr>
<td>GETMAXLPT</td>
<td>This value returns the maximum LPT port identifier supported by the system; this value ranges from 0x80 to 0xFF, such that 0x80 corresponds to LPT1, 0x81 to LPT2, 0x82 to LPT3, and so on.</td>
</tr>
<tr>
<td>RESETDEV</td>
<td>This value resets the printer device if the idComDev parameter specifies an LPT port; no function is performed if idComDev specifies a COM port.</td>
</tr>
<tr>
<td>SETDTR</td>
<td>This value sends the DTR (data-terminal-ready) signal.</td>
</tr>
<tr>
<td>SETRTS</td>
<td>This value sends the RTS (request-to-send) signal.</td>
</tr>
<tr>
<td>SETXOFF</td>
<td>This value causes transmission to act as if an XOFF character has been received.</td>
</tr>
<tr>
<td>SETXON</td>
<td>This value causes transmission to act as if an XON character has been received.</td>
</tr>
</tbody>
</table>
372.3 Returns
If the function successful, the return value is zero. Otherwise, the value is less than zero.

372.4 Errors
None.

372.5 Cross-References
None.

373 FlushComm
373.1 Synopsis
```c
typedef struct tagCOMSTAT {
    BYTE status;
    UINT cbInQue;
    UINT cbOutQue;
} COMSTAT;

int FlushComm(int NumComDev, int idQueue);
```

373.2 Description
The `FlushComm()` function flushes all characters from either the transmission or receiving queue of the device identified by the `NumComDev` parameter. `NumComDev`, which is returned by the `OpenComm()` function, identifies the communication device to be flushed. The `idQueue` parameter identifies which queue is to be flushed. If the value of the `idQueue` parameter is zero, the transmission queue is flushed. If the value is 1, the receiving queue is flushed.

373.3 Returns
If the function is successful, it returns zero. If the `NumComDev` parameter is not a valid communication device, or if the `idQueue` parameter is not a valid queue, the function returns a value less than zero. The function returns a value greater than zero if there is an error for the communication device identified by `NumComDev`.

373.4 Errors
The list of all possible error values greater than zero returned by this function are listed below.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE_BREAK</td>
<td>The hardware detected a break condition.</td>
</tr>
<tr>
<td>CE_CTSTO</td>
<td>A CTS (clear-to-send) timeout occurred; while a character was being transmitted, CTS was low for the duration specified by the <code>fCtsHold</code> member of the <code>COMSTAT</code> structure.</td>
</tr>
<tr>
<td>CE_DNS</td>
<td>A parallel device was not selected.</td>
</tr>
<tr>
<td>CE_OOP</td>
<td>A parallel device signaled that it is out of paper.</td>
</tr>
<tr>
<td>CE_OVERRUN</td>
<td>A character was not read from the hardware before the next character arrived; the character was lost.</td>
</tr>
<tr>
<td>CE_DSRTO</td>
<td>A DSR (data-set-ready) timeout occurred; while a character was being transmitted, DSR was low for the duration specified by the <code>fDsrHold</code> member of <code>COMSTAT</code>.</td>
</tr>
<tr>
<td>CE_FRAME</td>
<td>The hardware detected a framing error.</td>
</tr>
<tr>
<td>CE_IOE</td>
<td>An I/O error occurred during an attempt to communicate with a parallel device.</td>
</tr>
<tr>
<td>CE_MODE</td>
<td>The requested mode is not supported, or the <code>NumComDev</code> parameter is invalid if set, CE_MODE is the only valid error.</td>
</tr>
<tr>
<td>CE_MODE</td>
<td>The requested mode is not supported, or the <code>NumComDev</code> parameter is invalid if set, CE_MODE is the only valid error.</td>
</tr>
<tr>
<td>CE_MODE</td>
<td>The requested mode is not supported, or the <code>NumComDev</code> parameter is invalid if set, CE_MODE is the only valid error.</td>
</tr>
</tbody>
</table>
CE_PTO A timeout occurred during an attempt to communicate with a parallel device.

CE_RLSDTO An RLSD (receive-line-signal-detect) timeout occurred; while a character was being transmitted, RLSD was low for the duration specified by the fRLsdHold member of COMSTAT.

CE_RXOVER The receiving queue overflowed; there was either no room in the input queue or a character was received after the end-of-file character was received.

CE_RXPARI The hardware detected a parity error.

CE_TXFULL The transmission queue was full when a function attempted to queue a character.

373.5 Cross-References
GetCommError(), OpenComm()

374 GetCommError
374.1 Synopsis
typedef struct tagCOMSTAT {
    BYTE status;
    UINT cbInQue;
    UINT cbOutQue;
} COMSTAT;

int GetCommError(int NumComDev, COMSTAT *devStat);

374.2 Description
The GetCommError() function returns the most recent error value for the communication device specified by the NumComDev parameter. It also returns the current status for the communication device specified by the NumComDev parameter. When an error occurs, the system locks the communication port until the error can be cleared by this function. The NumComDev parameter identifies the communication device to be examined. The devStat parameter is a pointer to the COMSTAT structure, which receives the status of the communication device. If this parameter is set to NULL, then only the error values are returned by this function.

374.3 Returns
If the function is successful, the return value indicates the error value for the most recent communication function call that was made to the device identified by the NumComDev parameter.

374.4 Errors
The return value for the function can be a combination of the following values:

CE_BREAK The hardware detected a break condition.

CE_CTSSTO A CTS (clear-to-send) timeout occurred; while a character is transmitted, CTS is low for the duration specified by the fCtsHold member of the COMSTAT structure.

CE_DNS A parallel device was not selected.

CE_DSRTO A DSR (data-set-ready) timeout occurred; while a character is transmitted, DSR is low for the duration specified by the fDsrHold member of COMSTAT.

CE_FRAME The hardware detected a framing error.

CE_IOE An I/O error occurred during an attempt to communicate with a parallel device.
### CE_MODE
The requested mode is not supported, or the `idComDev` parameter is invalid; if set, CE_MODE is the only valid error.

### CE_OOP
A parallel device signaled that it is out of paper.

### CE_OVERRUN
A character was not read from the hardware before the next character arrived; the character was lost.

### CE_PTO
A timeout occurred during an attempt to communicate with a parallel device.

### CE_RLSDTO
An RLSD (receive-line-signal-detect) timeout occurred; while a character is being transmitted, RLSD is low for the duration specified by the `fRlsdHold` member of `COMSTAT`.

### CE_RXOVER
The receiving queue overflowed; there is either no room in the input queue or a character is received after the end-of-file character is received.

### CE_RXPARITY
The hardware detected a parity error.

### CE_TXFULL
The transmission queue was full when a function attempted to queue a character.

### 374.5 Cross-References
`OpenComm()`

#### 375 GetCommEventMask, SetCommEventMask

##### 375.1 Synopsis
```c
UINT GetCommEventMask(int NumComDev, int idEvtClear);
UINT *SetCommEventMask(int NumComDev, int idEvtMask);
```

##### 375.2 Description
The `GetCommEventMask()` function clears the event word of a communications device after retrieving this value. `NumComDev` identifies the communication device being examined. `NumComDev` is returned by the `OpenComm()` function. The `idEvtClear` parameter identifies the events cleared in the event word. The list of event values are as outlined below in the function `SetCommEventMask()`. The application calls `SetCommEventMask()` to enable the event before this function can record the occurrence of an event. Where the communication device event is a line status or printer error, then the application should call `GetCommEventMask()` before calling `GetCommError()` to retrieve the error.

`SetCommEventMask()` enables the event word of the specified communications device. `NumComDev` identifies the communication device to be enabled. This parameter is returned by the `OpenComm()` function. The `idEvtMask` parameter identifies the events to be enabled. This parameter can have any of the following values:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV_BREAK</td>
<td>This value is set when a break is detected on input.</td>
</tr>
<tr>
<td>EV_CTS</td>
<td>This value is set when the CTS (clear-to-send) signal changes state.</td>
</tr>
<tr>
<td>EV_CTSS</td>
<td>This value is set to indicate the current state of the CTS signal.</td>
</tr>
<tr>
<td>EV_DSR</td>
<td>This value is set when the DSR (data-set-ready) signal changes state.</td>
</tr>
<tr>
<td>EV_ERR</td>
<td>This value is set when a line-status error occurs; line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.</td>
</tr>
<tr>
<td>EV_PERR</td>
<td>This value is set when a printer error is detected on a parallel device; errors are CE_DNS, CE_IOE, CE_LOOP, and CE_PTO.</td>
</tr>
<tr>
<td>EV_RING</td>
<td>This value is set to indicate the state of ring indicator during the last modem interrupt.</td>
</tr>
<tr>
<td>EV_RLSD</td>
<td>This value is set when the RLSD (receive-line-signal-detect) signal changes state.</td>
</tr>
</tbody>
</table>
EV_RLSDS This value is set to indicate the current state of the RLSD signal.

EV_RXCHAR This value is set when any character is received and placed in the receiving queue.

EV_RXFLAG This value is set when the event character is received and placed in the receiving queue; the event character is specified in the device’s control block.

EV_TXEMPTY This value is set when the last character in the transmission queue is sent.

375.3 Returns

If `GetCommEventMask()` is successful, its return value is the current event-word for the communications device specified by `NumComDev`. The bits in the event-word identifies whether a given event has occurred or not. The bit is set to 1, if the event has occurred.

If `SetCommEventMask()` is successful, it returns a pointer to the event word for the communication device identified by the `NumComDev` parameter. The bits in the event word indicate whether a given event has occurred or not. A bit is set to 1, if the event has occurred.

It should be noted that only events that are enabled are recorded. The function `GetCommEventMask()` clears the event word after retrieving it.

375.4 Errors

None.

375.5 Cross-References

`GetCommError()`, `OpenComm()`

376 GetCommState, SetCommState

376.1 Synopsis

```c
#include <sys/param.h>

int GetCommState(int NumComDev, DCB *iddcb);
int SetCommState(DCB *iddcb);
```

376.2 Description

The `GetCommState()` function gets the device control block for the device identified by the `NumComDev` parameter. The `NumComDev` parameter identifies the communication device. This parameter is returned by the `OpenComm()` function. The `iddcb` parameter contains a pointer to the `DCB` (Device Control Block) structure that defines the different control settings for the communication device identified by `NumComDev`.

The `SetCommState()` function is used to set the communications device to the state specified by the settings in the `DCB` structure. The `iddcb` parameter contains a pointer to the `DCB` structure, which defines the different control settings for the communication device. It should be noted that the `id` member of this structure should specify the device.

This function reinitializes the hardware and controls of the communication device as identified by the `DCB` structure but does not empty the transmission or receiving queues.

376.3 Returns

If `GetCommState()` is successful, it returns zero. Otherwise, it returns a value of less than zero.

If `SetCommState()` is successful, it returns zero. Otherwise, it returns a value of less than zero.

376.4 Errors

None.

376.5 Cross-References

`OpenComm()`
377  ReadComm, WriteComm

377.1  Synopsis

int ReadComm(int NumComDev, void *lpdBuf, int nbRead);

int WriteComm(int NumComDev, void *lpdBuf, int nbWrite);

377.2  Description

The ReadComm() function reads up to an indicated number of bytes from the communication device specified by the NumComDev parameter. The NumComDev parameter, which is returned by the OpenComm() function, identifies the communication device. The lpdBuf parameter contains a pointer to the buffer that contains the bytes to be read. The nbRead parameter identifies the number of bytes that are read from the buffer.

The WriteComm() function writes up to an indicated number of bytes to the communication device specified by the NumComDev parameter. The NumComDev parameter, which is returned by the OpenComm() function, identifies the communication device. The lpdBuf parameter contains a pointer to the buffer containing the bytes to be written. The nbWrite parameter identifies the number of bytes to be written to the buffer.

377.3  Returns

If the function ReadComm() is successful, the return value contains the number of bytes that were read. If the function is unsuccessful, the return value is less than zero and its absolute value identifies the number of bytes that were read. If the communication device is a parallel port, the return value is always zero. It is good practice to call the GetCommError() function to check the error state even though the return value is zero, since errors can occur when the number of bytes is zero. When an error occurs, the function GetCommError() is called to retrieve the error state. If the return value is zero, there are no bytes present. If the return value is less than the nbRead parameter, the number of bytes in the receiving queue is less than that specified by this parameter. If the return value is equal to nbRead, there may be additional bytes that are queued for the device.

If the WriteComm() function is successful, the return value contains the number of bytes written. If the function is unsuccessful, the return value is less than zero and its absolute value identifies the number of bytes written. In case of an error, the GetCommError() function should be used to retrieve the error state.

If the communication device is a serial port then the WriteComm() function deletes the data in the transmission queue, if there is not enough room in the queue for the additional bytes. It is a good practice to check the available space in the queue by calling the GetCommError() function before calling the WriteComm() function. The size of the transmission queue that is set when the OpenComm() function is called, should be larger or at least equal to the size of the largest expected output string that will be written.

377.4  Errors

None.

377.5  Cross-References

GetCommError(), OpenComm(), TransmitCommChar()

378  TransmitCommChar, UngetCommChar

378.1  Synopsis

int TransmitCommChar(int NumComDev, char TransCh);

int UngetCommChar(int NumComDev, char UngetCh);

378.2  Description

The TransmitCommChar() function puts the specified character contained in the TransCh parameter at the head of the transmission queue of the device identified by the NumComDev parameter. The NumComDev parameter returned by the OpenComm() function, identifies the communication device. The TransCh parameter contains the character to be transmitted. This function cannot be called repeatedly if the device specified by NumComDev is not transmitting. If the function has placed a character in the transmission queue, this character should be transmitted before the function is called again. This is done by checking the return value of this function.
The `UngetCommChar()` function puts the specified character contained in `UngetCh` back in the receiving queue of the device identified by the `NumComDev` parameter. The `NumComDev` parameter returned by the `OpenComm()` function identifies the communication device. The `UngetCh` parameter contains the character that is placed in the receiving queue to be read back when the next read operation takes place. Once this function places a character in the receiving queue, a read has to happen before this function can be called again.

### 378.3 Returns

The function `TransmitCommChar()` returns zero, if it is successful. The return value is less than zero if the character could not be transmitted.

`UngetCommChar()` returns zero, if it is successful. Otherwise, the return value is less than zero.

### 378.4 Errors

None.

### 378.5 Cross-References

`OpenComm()`

### 379 GetDriveType

#### 379.1 Synopsis

`UINT GetDriveType(int nDriveNumber);`

#### 379.2 Description

The `GetDriveType()` function reports whether the drive specified in the `DriveNumber` parameter is removable, fixed, or remote. The `DriveNumber` parameter of value 0 is considered to specify drive A, a value of 1 specifies drive B, and so on.

#### 379.2 Returns

- `DRIVE_REMOVABLE` removable media (for example, floppy disk drives)
- `DRIVE_FIXED` fixed media (for example, hard disk drives)
- `DRIVE_REMOTE` network drives

#### 379.4 Errors

None.

#### 379.5 Cross-References

None.

### 380 GetSystemDirectory

#### 380.1 Synopsis

`UINT GetSystemDirectory(LPSTR lpBuffer, UINT nSize);`

#### 380.2 Description

The `GetSystemDirectory()` function retrieves the system directory that contains drivers, libraries, font files, and so on.

The `lpBuffer` parameter points to a buffer which will contain a null-terminated string specifying the path to the system directory. Only `nSize` bytes will be copied to this buffer. The recommended minimum value for `nSize` is 144 bytes.

#### 380.3 Returns

`GetSystemDirectory()` returns the number of bytes required (excluding the null-terminator) to store the full pathname to the System directory, if it is successful.
### 380.4 Errors
None.

### 380.5 Cross-References
*GetWindowsDirectory()*

---

### 381 GetTempDrive

#### 381.1 Synopsis
BYTE GetTempDrive(char cDriveLetter);

#### 381.2 Description
The `GetTempDrive()` function returns a drive letter that can be used as temporary space. The `cDriveLetter` parameter is currently ignored.

#### 381.3 Returns
The letter returned can range from A-Z. The case is not guaranteed. The drive letter A is associated with drive number 0, B with 1, and so on. If no drive letters are available, this function returns the letter of the current drive.

#### 381.4 Errors
None.

#### 381.5 Cross-References
None.

---

### 382 GetTempFileName

#### 382.1 Synopsis
```
int GetTempFileName(BYTE cDriveLetter, LPCSTR lpPrefixString, UINT uUnique,
                   LPSTR lpTempFilename);
```

#### 382.2 Description
The `GetTempFileName()` function creates a file name that can be used for temporary storage. These are the parameters associated with the `GetTempFileName()` function.

- **cDriveLetter**: This parameter suggests a drive number for the temporary file to reside; if zero, the default (current) drive is used.
- **lpPrefixString**: This parameter is a pointer to a NULL-terminated string to be used as the prefix to the filename; the string must consist of OEM-defined characters.
- **uUnique**: If non-zero, this number will be appended to the temporary filename; otherwise, the current system time will be used.
- **lpTempFilename**: This parameter is a pointer to a buffer, where the temporary filename will be stored; this should be at least 144 bytes long. The application should expect the filename to consist of only OEM-defined characters.

The file returned by `GetTempFileName()` is not deleted when the application exits. It is the application's responsibility to remove the file on exit.

To avoid problems with OEM character strings and the system strings, `_lopen()` and `_lclose()` should be used.

The following is the order of precedence (from highest to lowest) in which the drive letter is determined:

- TEMP environment variable
- a local fixed disk
- the `cDriveLetter` parameter
If the `uUnique` parameter is zero, the function constructs a unique name for the temporary file, and thus attempts to create the file. If the file already exists, it increments the unique identifier value and tries again. After it succeeds in finding the filename, it closes the file and returns.

### 382.3 Returns

This function returns the unique number that is used to create the filename, if the `uUnique` parameter is zero. Otherwise, `GetTempFileName()` returns the `uUnique` parameter. The `lpszTempFileName` parameter contains a string of the form:

```
d:\path\prefixuuu.tmp
```

where

- `d:` This value is a drive letter on which the temporary file will reside.
- `path` This value is a path to the directory containing the temporary file; this is either the system directory (see `GetWindowsDirectory()`) or the value of the TEMP environment variable.
- `prefix` This value is a prefix to append to the file name. All letters of the string are used, however, `prefix` is no longer than 3 letters.
- `uuu` This value is a hexadecimal value of `uUnique`, or a unique number based on the system clock.

### 382.4 Errors

None.

### 382.5 Cross-References

`lopen()`, `lclose()`, `GetWindowsDirectory()`

---

### 383 GetWindowsDirectory

#### 383.1 Synopsis

```
UINT GetWindowsDirectory(LPSTR lpBuffer, UINT nSize);
```

#### 383.2 Description

The `GetWindowsDirectory()` function returns the path to the system directory, which contains the application's .INI files, temporary files, and so on.

The `lpBuffer` parameter is a pointer to a buffer that receives the path name. No more than `nSize` bytes are copied to this buffer. A size of 144 bytes is the recommended minimum size for this buffer.

#### 383.3 Returns

This function returns the number of bytes required to hold the entire path name (excluding the null-terminator).

#### 383.4 Errors

None.

#### 383.5 Cross-References

None.

---

### 384 OpenFile

#### 384.1 Synopsis

```
HFILE OpenFile(LPCSTR lpFileName, OFSTRUCT *lpOfs, UINT wMode);
```

#### 384.2 Description

The `OpenFile()` function creates, opens, reopens, or deletes a file. The following table describes the function's parameters.
**lpFileName**  This value is a pointer to a null-terminated string to the filename.

**lpOfs**  This value is a pointer to structure that contains information about the opened file; that structure can then be used in subsequent calls to OpenFile() to refer to the opened file.

**wMode**  This value specifies any special actions taken as well as the attributes of the file.

Values for **wMode** can be a combination of the following flags:

- **OFCancelar**  When used in conjunction with OF_PROMPT, this flag adds a Cancel button to the dialog box; pressing Cancel causes OpenFile() to return a file-not-found error.

- **OF_CREATE**  This flag creates a new file, or truncates the file if the file already exists. Sharing flags are ignored when this flag is present; if sharing options are required, the file is closed and reopened with the proper parameters.

- **OF_DELETE**  This flag deletes the file.

- **OF_EXIST**  This flag checks to see if the file exists; file and date/time stamp are not modified.

- **OF_PARSE**  This flag fills the lpOfs structure; no other action is performed.

- **OF_PROMPT**  This flag displays a dialog box if the requested file does not exist and prompts the user to insert the disk containing the file in drive A.

- **OF_READ**  This flag opens file for read only.

- **OF_READWRITE**  This flag opens file for read and write.

- **OF_REOPEN**  This flag reopens file with new parameters.

- **OF_SEARCH**  This flag searches in directories specified in the path as well as the Windows and System directories (see GetWindowsDirectory() and GetSystemDirectory()), even when given a full path.

- **OF_SHARE_COMPAT**  This flag opens the file in compatibility mode; any other program can open the file any number of times. OpenFile() fails if it has been opened with any other sharing options.

- **OF_SHARE_DENY_NONE**  This flag opens the file and allows any other program to open the file for reading or writing. OpenFile() fails if it has already been opened in compatibility mode (OF_SHARE_COMPAT) or read-only mode by any other program.

- **OF_SHARE_DENY_READ**  This flag opens the file and denies read access by any other program. OpenFile() fails if it has already been opened in compatibility mode or read access by any other program.

- **OF_SHARE_DENY_WRITE**  This flag opens the file and denies write access by any other program. OpenFile() fails if it has already been opened in compatibility mode or write access by any other program.

- **OF_SHARE_EXCLUSIVE**  This flag opens the file with exclusive mode; it denies read or writes to the file by all other programs. OpenFile() fails if the file has already been opened for read or write access by any program, including the current one.

- **OF_VERIFY**  This flag compares the time and date in lpOfs with the one in the specified file. OpenFile() returns HFILE_ERROR if the dates and times do not match.

- **OF_WRITE**  This flag opens the file for writing only.

### 384.3 Returns

The OpenFile() function returns a file handle that can be used with the standard C libraries, if it is successful.
Note: This handle may not be valid. In particular, the OF_EXIST and OF_DELETE functions return a meaningless value.

384.4 Errors
None.

384.5 Cross-References
None.

385 SetHandleCount

385.1 Synopsis
UINT SetHandleCount(UINT nHandles);

385.2 Description
The SetHandleCount() function sets the number of file handles available to an application.

The nHandles parameter sets the number of file handles available to an application. This value cannot be greater than 255.

385.3 Returns
This function returns the number of file handles available to the application, if it is successful.

385.4 Errors
None.

385.5 Cross-References
None.

386 _lclose

386.1 Synopsis
HFILE _lclose(HFILE FileHandle);

386.2 Description
The _lclose() function closes the file described by the file handle, FileHandle, which is of type HFILE. By closing a file, described by FileHandle, with the _lclose() function, the file becomes unusable for further read/write activities until it is reopened.

386.3 Returns
If the function is successful, _lclose() returns a value of zero. Otherwise, it will return a value of HFILE_ERROR.

386.4 Errors
None.

386.5 Cross-References
_lopen()

387 _lread

387.1 Synopsis
UINT _lread(HFILE hFile, const void *BufferPtr, UINT NumBytes);

387.2 Description
The _lread() function reads a specified number of bytes from a file into memory. The function supports objects that are larger than 64K. The hFile parameter contains a handle to an open file. The BufferPtr parameter is a pointer to a
memory buffer that will be used to store the data read from the file. The NumBytes parameter specifies the number of bytes to read from the file.

387.3 Returns
If the function is successful, it returns the number of bytes read from the file. If the function encounters a file reading error other than the an end-of-file (EOF) error, it returns HFILE_ERROR.

387.4 Errors
None.

387.5 Cross-References
None.

388 _lcreat

388.1 Synopsis
HFILE _lcreat(LPCSTR FileName, int FileAttr);

388.2 Description
The _lcreat() function opens a file, described by FileName, for reading and/or writing. If the file to be opened does not exist, _lcreat() will attempt to create the file first. The FileAttr parameter is used to describe how the file is to be opened and/or created by _lcreat(). The values valid for FileAttr are the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This value indicates a normal file; this file can be read and written to by anyone.</td>
</tr>
<tr>
<td>1</td>
<td>This value indicates a read-only file; this file can only be written to and cannot be opened for writing.</td>
</tr>
<tr>
<td>2</td>
<td>This value indicates a hidden file; this file has the hidden attribute and will not show up in directory listings.</td>
</tr>
<tr>
<td>3</td>
<td>This value indicates a system file; this file is a system file and will not show up in directory listings.</td>
</tr>
</tbody>
</table>

388.3 Returns
If _lcreat() is successful, it returns a file handle to the file newly created or opened. If there is an error in opening or creating the file, an error value of HFILE_ERROR is returned.

388.4 Errors
None.

388.5 Cross-References
None.

389 _llseek

389.1 Synopsis
LONG _llseek(HFILE FileHandle, LONG OffsetFromCurrent, int StartPos);

389.2 Description
The _llseek() function moves the current file position pointer of the file described by FileHandle (of type HFILE), an offset of OffsetFromCurrent from the position described by StartPos(). StartPos() contains a value describing the place in the file from which to begin offsetting. StartPos() can have the following values:
The return value of _llseek() is the final offset, from the beginning of the file, to which the pointer is now aimed. If the function was unsuccessful in completing the offset, the return value is HFILE_ERROR.

Errors
None.

Cross-References
None.

_lopen

Synopsis
HFILE _lopen(LPCSTR FileName, int FileMode);

Description
The _lopen() function is used to open up a file as described by FileName, with the open options as described by FileMode. FileMode can have the following values:

REQUIRED:
READ File is opened for read access only.
READ_WRITE File is opened for read/write access.
WRITE File is opened for write access only.

OPTIONAL:
OF_SHARE_COMPAT Compatibility mode allows any process to open the file as many times as they want; an error occurs when the file is opened with any other share flag.
OF_SHARE_DENY_NONE Do not deny read/write access to any other process; an error occurs if the file was opened in compatibility mode.
OF_SHARE_DENY_READ Deny read access to any other process; an error occurs if the file was opened in compatibility mode or any other mode allowing read access.
OF_SHARE_DENY_WRITE Deny write access to any other process; an error occurs if the file was opened in compatibility mode or any other mode allowing write access.
OF_SHARE_EXCLUSIVE Deny access completely to any other process; an error occurs if the file was opened in any mode by any other process.

Returns
If _lopen() is successful, a file handle to the newly opened file is returned. If the function is unsuccessful, a HFILE_ERROR is returned.

Errors
None.

Cross-References
None.
391 _lwrite

391.1 Synopsis
UINT _lwrite(HFILE hFile, const void *BufferPtr, UINT NumBytes);

391.2 Description
The _lwrite() function writes a specified number of bytes of memory to a file. The function supports objects that are larger than 64K. The hFile parameter contains a handle to an open file. The BufferPtr parameter is a pointer to a memory buffer that contains the bytes of data to write to the file. The NumBytes parameter specifies the number of bytes in the memory buffer to write to the file. If the value of the NumBytes parameter is zero, the function will expand or truncate the file to the current file pointer position.

391.3 Returns
If the function is successful, it returns the number of bytes written to the file. If the function is not successful, it returns an error value of HFILE_ERROR.

391.4 Errors
None.

391.5 Cross-References
None.

392 RegCloseKey

392.1 Synopsis
LONG RegCloseKey (HKEY hkey);

392.2 Description
The RegCloseKey() function releases the handle of the key specified by the hkey parameter by closing the key. The Registration Database is updated when all keys are closed.

392.3 Returns
If successful, the function returns ERROR_SUCCESS. Otherwise, it returns an error value.

392.4 Errors
None.

392.5 Cross-References
None.

393 RegCreateKey, RegOpenKey

393.1 Synopsis
LONG RegCreateKey(HKEY hkey, LPCSTR szSubKey, HKEY *lpResult);
LONG RegOpenKey(HKEY hkey, LPCSTR szSubKey, HKEY *lpResult);

393.2 Description
The RegCreateKey() function either creates a Registration Database key or opens the specified key if it already exists. The szSubKey parameter points to the string that specifies the name of the key to open or create. The lpResult parameter is the address of the handle of the key created or opened. The hkey parameter is the handle of the parent key which can be HKEY_CLASSES_ROOT. It cannot be NULL.

The RegOpenKey() function opens a Registration Database key. The szSubKey parameter points to the string that specifies the name of the key to open. The lpResult parameter is the address of the handle of the key opened. The hkey parameter is the handle of the parent key which may be HKEY_CLASSES_ROOT and cannot be NULL.
393.3 Returns
The `RegCreateKey()` function returns ERROR_SUCCESS, if it is successful. Otherwise, it returns an error value.

The `RegOpenKey()` function returns ERROR_SUCCESS, if it is successful. Otherwise, it returns an error value.

393.4 Errors
None.

393.5 Cross-References
None.

---

394 RegDeleteKey

394.1 Synopsis
LONG RegDeleteKey(HKEY hkey, LPCSTR szSubKey);

394.2 Description
The `RegDeleteKey()` function deletes the Registration Database subkey specified by the `szSubKey` parameter. The `hkey` parameter defines the handle of the key whose subkey is to be deleted. The `hkey` parameter can be HKEY_CLASSES_ROOT.

394.3 Returns
If successful, `RegDeleteKey()` returns ERROR_SUCCESS. Otherwise, it returns an error value.

394.4 Errors
`RegDeleteKey()` returns an error value if it fails. If the returned error is ERROR_ACCESS_DENIED, either the application does not have the permission to delete the specified subkey or another application has the specified subkey open.

394.5 Cross-References
None.

---

395 RegEnumKey

395.1 Synopsis
LONG RegEnumKey(HKEY hkey, DWORD iSubKey, LPSTR szBuffer, DWORD cbBuffer);

395.2 Description
The `RegEnumKey()` function enumerates the subkeys of the Registration Database entry specified by the `hkey` parameter, an open handle (which can be HKEY_CLASSES_ROOT). The `iSubKey` parameter is the index of the subkey to be retrieved. It should be set to zero the first time `RegEnumKey()` is called. The `szBuffer` parameter is a buffer of size `cbBuffer` into which the name of the subkey is copied by `RegEnumKey()`.

395.3 Returns
If successful, `RegEnumKey()` returns ERROR_SUCCESS. Otherwise it returns an error value.

395.4 Errors
None.

395.5 Cross-References
None.
RegQueryValue, RegSetValue

Synopsis
LONG RegQueryValue(HKEY hkey, LPCSTR szSubKey, LPSTR szValue, LONG *lpcb);
LONG RegSetValue(HKEY hkey, LPCSTR szSubKey, DWORD fdwType, LPCSTR szValue,
                  DWORD cb);

Description
RegQueryValue() queries the Registration Database and returns the value of szSubKey. The szSubKey parameter is a child of the Registration Database entry whose handle is hkey. RegQueryValue() returns the value in the szValue buffer, and the length of the value string in the lpcb parameter.

RegSetValue() sets the subkey specified by szSubKey to a value stored in szValue. If the szSubKey parameter is NULL or is a pointer to an empty string, RegSetValue() sets the value of the hkey parameter. The hkey parameter is the non-NULL handle of the key whose subkey is to be modified. The fdwType parameter must be set to REG_SZ for Windows 3.1. The cb parameter is the size of szValue in bytes. It is ignored by RegQueryValue() in Windows 3.1.

Returns
RegQueryValue() returns ERROR_SUCCESS, if it is successful.
RegSetValue() returns ERROR_SUCCESS, if it is successful.

Errors
None.

Cross-References
None.

IsBadCodePtr

Synopsis
BOOL IsBadCodePtr(FARPROC FunctPtr);

Description
The IsBadCodePtr() function validates the given pointer to executable code. The FunctPtr parameter points to the entry point of a function.

Returns
If the pointer is correct, the IsBadCodePtr() function returns FALSE. If the pointer is incorrect, the IsBadCodePtr() function returns TRUE.

Errors
None.

Cross-References
IsBadHugeReadPtr(), IsBadHugeWritePtr(), IsBadReadPtr(), IsBadStringPtr(), IsBadWritePtr()

IsBadHugeReadPtr

Synopsis
BOOL IsBadHugeReadPtr(const void _huge *MemPtr, DWORD Size);

Description
The IsBadHugeReadPtr() function determines whether a huge pointer to readable memory is valid. The MemPtr parameter is a huge pointer to the first byte of the readable memory block. The block size can be bigger than 64k. The Size parameter is the total number of bytes in the memory block.
If the pointer is correct, the `IsBadHugeReadPtr()` function returns FALSE. If the pointer is incorrect, the `IsBadHugeReadPtr()` function returns TRUE.

## Errors
None.

### Cross-References
`IsBadCodePtr()`, `IsBadHugeWritePtr()`, `IsBadReadPtr()`, `IsBadStringPtr()`, `IsBadWritePtr()`

### Synopsis
```c
BOOL IsBadHugeWritePtr(void _huge* MemPtr, DWORD Size);
```

### Description
The `IsBadHugeWritePtr()` function validates the given huge pointer to a writable memory block. The `MemPtr` parameter is a huge pointer to the first byte of the writable block. The block size can be bigger than 64k. The `Size` parameter is the total number of bytes in the memory block.

### Returns
If the pointer is correct, the `IsBadHugeWritePtr()` function returns FALSE. If the pointer is incorrect, the `IsBadHugeWritePtr()` function returns TRUE.

### Errors
None.

### Cross-References
`IsBadCodePtr()`, `IsBadHugeReadPtr()`, `IsBadReadPtr()`, `IsBadStringPtr()`, `IsBadWritePtr()`

### Synopsis
```c
BOOL IsBadReadPtr(const void *MemPtr, UINT Size);
```

### Description
The `IsBadReadPtr()` function validates the given pointer to readable memory. The `MemPtr` parameter is a pointer to the first byte of the readable memory block. The `Size` parameter is the total number of bytes in the memory block.

### Returns
If the pointer is correct, the `IsBadReadPtr()` function returns FALSE. If the pointer is incorrect, the `IsBadReadPtr()` function returns TRUE.

### Errors
None.

### Cross-References
`IsBadCodePtr()`, `IsBadHugeReadPtr()`, `IsBadReadPtr()`, `IsBadStringPtr()`, `IsBadWritePtr()`

### Synopsis
```c
BOOL IsBadStringPtr(const void *MemPtr, UINT Size);
```

### Description
The `IsBadStringPtr()` function validates the given pointer to a readable string. The `MemPtr` parameter is a pointer to the first byte of the readable string. The `Size` parameter is the total number of bytes in the memory block.

### Returns
If the pointer is correct, the `IsBadStringPtr()` function returns FALSE. If the pointer is incorrect, the `IsBadStringPtr()` function returns TRUE.

### Errors
None.

### Cross-References
`IsBadCodePtr()`, `IsBadHugeReadPtr()`, `IsBadReadPtr()`, `IsBadStringPtr()`, `IsBadWritePtr()`
401.2 Description
The IsBadStringPtr() function validates the given pointer to a string. The MemPtr parameter is the pointer to the first byte of the string.

Note: The string is null-terminated.

The Size parameter is the total number of bytes in the string.

401.3 Returns
If the pointer is correct, the IsBadStringPtr() function returns FALSE. If the pointer is incorrect, the IsBadStringPtr() function returns TRUE.

401.4 Errors
None.

401.5 Cross-References
IsBadCodePtr(), IsBadHugeReadPtr(), IsBadHugeWritePtr(), IsBadReadPtr(), IsBadWritePtr()

402 IsBadWritePtr

402.1 Synopsis
BOOL IsBadWritePtr(void *MemPtr, UINT Size);

402.2 Description
The IsBadWritePtr() function validates a given pointer to writable memory. The MemPtr parameter is the pointer to the first byte of the writable block. The Size parameter is the total number of bytes in the memory block.

402.3 Returns
If the pointer is correct, the IsBadWritePtr() function returns FALSE. If the pointer is incorrect, the IsBadWritePtr() function returns TRUE.

402.4 Errors
None.

402.5 Cross-References
IsBadCodePtr(), IsBadHugeReadPtr(), IsBadHugeWritePtr(), IsBadReadPtr(), IsBadWritePtr()
Section 5 - Application Support Functions

403 ExtractIcon

403.1 Synopsis
HICON ExtractIcon (HINSTANCE hinst, LPCSTR szBinary, UINT iIcon)

403.2 Description
The ExtractIcon() function returns a handle to an icon stored inside the szBinary executable, DLL, or icon file. The application that calls the functions is identified by the hinst parameter. The hIcon parameter specifies the index of the icon to be retrieved. If the value of hIcon is zero, the handle to the first icon is returned. If the value is -1, the total number of icons in the file is returned.

403.3 Returns
If successful, ExtractIcon() returns the handle to the requested icon. It returns the NULL value if the specified file contains no icons. ExtractIcon() returns 1 if the specified file in not an executable file, DLL, or icon file.

403.4 Errors
None.

403.5 Cross-References
None.

404 FindExecutable

404.1 Synopsis
HINSTANCE FindExecutable (LPCSTR lpszFile, LPCSTR lpszDir, LPCSTR lpszResult)

404.2 Description
FindExecutable() finds and retrieves the executable filename that is associated with a specified filename. The lpszFile parameter points to a null-terminated string that specifies a filename (which can be a document or an executable file). The lpszDir parameter points to a null-terminated string that specifies a full directory path for the default directory. The lpszResult parameter points to a null-terminated string of an executable file.

404.3 Returns
If successful, FindExecutable() returns a value greater than 32.

404.4 Errors
If unsuccessful, FindExecutable() returns one of the following error codes:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The system was out of memory, executable file was corrupt, or relocations were invalid.</td>
</tr>
<tr>
<td>2</td>
<td>The file was not found.</td>
</tr>
<tr>
<td>3</td>
<td>The path was not found.</td>
</tr>
<tr>
<td>5</td>
<td>An attempt was made to dynamically link to a task, or there was a sharing or network-protection error.</td>
</tr>
<tr>
<td>6</td>
<td>A library required separate data segments for each task.</td>
</tr>
<tr>
<td>8</td>
<td>There was insufficient memory to start the application.</td>
</tr>
<tr>
<td>10</td>
<td>The Windows version was incorrect.</td>
</tr>
<tr>
<td>11</td>
<td>An executable file was invalid. Either it was not a Windows application or there was an error in the .EXE image.</td>
</tr>
<tr>
<td>12</td>
<td>The application was designed for a different operating system.</td>
</tr>
<tr>
<td>13</td>
<td>The application was designed for MS-DOS 4.0.</td>
</tr>
<tr>
<td>14</td>
<td>The type of executable file was unknown.</td>
</tr>
<tr>
<td>15</td>
<td>An attempt was made to load a real-mode application (developed for an earlier version of Windows)</td>
</tr>
<tr>
<td>16</td>
<td>An attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read-only.</td>
</tr>
<tr>
<td>19</td>
<td>An attempt was made to load a compressed executable file. The file must be decompressed before it can be loaded.</td>
</tr>
<tr>
<td>20</td>
<td>A dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.</td>
</tr>
<tr>
<td>21</td>
<td>An application requires Microsoft Windows 32-bit extensions.</td>
</tr>
<tr>
<td>31</td>
<td>There is no association for the specified file type.</td>
</tr>
</tbody>
</table>

### Cross-References

None.

### GetPrivateProfileString, GetProfileString

#### Synopsis

```c
int GetPrivateProfileString(LPCSTR lpSect, LPCSTR lpKey, LPCSTR lpDefault, LPSTR lpReturn, int nSize, LPCSTR lpFile);

int GetProfileString(LPCSTR lpSect, LPCSTR lpKey, LPCSTR lpDefault, LPSTR lpReturn, int nSize);
```

#### Description

The `GetPrivateProfileString()` and `GetProfileString()` functions return a string of data from an initialization file. `GetProfileString()` is equivalent to the alternative `GetPrivateProfileString()` with the default windows initialization file, WIN.INI.

A system initialization file consists of lines of text broken into named sections consisting of a string of characters starting with the "[" character and ending with the "]" character, continuing to the next section delimited by the "[" and "]" characters. Lines starting with the "#" or ";" character are comments strings, as are blank lines and are not included in processing by the profile functions.

Data lines within a named section consist of a key string followed by the "=" character, followed by the key data. Section names and key strings are case insensitive, and any leading and trailing blanks removed before any processing occurs.

The profile functions parameters consist of a section name, without the leading and trailing braces, the desired key string and a default string to return if the key string cannot be found, a buffer to store the requested key string data is provided as well as a parameter specifying the length of the buffer. The default string is returned if the specified file cannot be found, the section does not exist or the key string cannot be found. The returned string is truncated to fit
into the user specified buffer if it too long. The default string may point to a zero length string which will have the
effect of copying an empty string to the users return buffer.

If no key string is provided, all key strings in the named section are returned separated by a NULL terminator, and
terminated by two NULL terminators. If the user specified return buffer is too small, as much data as possible is
copied including the two terminating NULL characters.

405.3 Returns
The profile functions return the number of characters copied into the return buffer, not including the terminating null
character. The user supplied buffer is filled in with the requested data, or the default string up to the size of the
buffer.

The functions will return default data in the event that the file cannot be found or read, or the section or key data
cannot be found.

405.4 Errors
None.

405.5 Cross-References
WritePrivateProfileString(), WriteProfileString()

406 WritePrivateProfileString, WriteProfileString

406.1 Synopsis
BOOL WritePrivateProfileString(LPCSTR lpSect, LPCSTR lpKey, LPCSTR lpData, LPCSTR lpFile);
BOOL WriteProfileString(LPCSTR lpSect, LPCSTR lpKey, LPCSTR lpData);

406.2 Description
The WritePrivateProfileString() and WriteProfileString() functions write out a key string and its associated data to
the requested section of the specified file, or the default win.ini file. If the file does not exist, it is created. If the
section does not exist, it is created. If the key data exists, it is overwritten.

A system initialization file consists of lines of text broken into named sections consisting of a string of characters
starting with the "[" character and ending with the "]" character, continuing to the next section delimited by the "[" and
"]" characters. Lines starting with the "#" or ";" character are comments strings as are blank lines and are not
included in processing by the profile functions.

Data lines within a named section consist of a key string followed by the "=" character, followed by the key data.
Section names and key strings are case insensitive and any leading and trailing blanks are removed before
processing occurs.

The write profile functions allow a given key string and its data to be deleted if the key string data supplied is
NULL. A named section can be deleted along with all its associated key strings and key data, if the key string is
NULL.

406.3 Returns
The function returns TRUE, if it is successful. If it fails because the file is not writable or cannot be found, it returns
FALSE.

406.4 Errors
None.

406.5 Cross-References
GetPrivateProfileString(), GetProfileString()

407 GetPrivateProfileInt, GetProfileInt

407.1 Synopsis
UINT GetPrivateProfileInt(LPCSTR lpSect, LPCSTR lpKey, int nDefault, LPCSTR lpFile);
UINT GetProfileInt(LPCSTR lSect, LPCSTR lpKey, int nDefault);

407.2 Description
The GetPrivateProfileInt() and GetProfileInt() functions return an integer value from the corresponding initialization file. A default value can be supplied in case the key string in the requested named section cannot be found. The key string data may be preceded with the "+" and "-" characters, or can be given in hexadecimal format. If the key string data is not a valid number, these functions return zero.

407.3 Returns
The return value is either an unsigned integer value retrieved from the specified initialization file or the default value supplied if the key string is not found in the appropriate section.

The functions will return default data in the event that the file cannot be found or read, or the section or key data cannot be found.

407.4 Errors
None.

407.5 Cross-References
GetPrivateProfileString(), GetProfileString()

408 AnsiLower, AnsiLowerBuff

408.1 Synopsis
LPSTR WINAPI AnsiLower(LPSTR lpszStr);
UINT WINAPI AnsiLowerBuff(LPSTR lpszString, UINT cbStr);

408.2 Description
The AnsiLower() and AnsiLowerBuff() functions convert character strings to lowercase. AnsiLower() converts all the characters in the zero-terminated string. If a single character is passed when the upper word is zero, the character is converted. AnsiLowerBuff() converts the number of characters specified by cbStr. If cbStr is zero, the length defaults to 65,536.

408.3 Returns
AnsiLower() returns a pointer to the converted character string. If unsuccessful, it returns a value that contains the converted character in the low byte of the low word. AnsiLowerBuff() returns the length of the converted string. If unsuccessful, it returns zero.

408.4 Errors
None.

408.5 Cross-References
AnsiUpper()

409 AnsiUpper, AnsiUpperBuff

409.1 Synopsis
LPSTR WINAPI AnsiLower(LPSTR lpszStr);
UINT WINAPI AnsiLowerBuff(LPSTR lpszString, UINT cbStr);

409.2 Description
The AnsiUpper() and AnsiUpperBuff() functions convert character strings to uppercase. AnsiUpper() converts all the characters in the zero-terminated string. If a single character is passed when the upper word is zero, the character is converted.

AnsiUpperBuff() converts the number of characters specified by cbStr. If cbStr is zero, the length defaults to 65,536.
409.3 Returns

AnsiUpper() returns a pointer to the converted character string. If unsuccessful, it returns a value that contains the converted character in the low byte of the low word. AnsiUpperBuff() returns the length of the converted string. If unsuccessful, it returns zero.

409.4 Errors

None.

409.5 Cross-References

AnsiLower()

410 AnsiNext, AnsiPrev

410.1 Synopsis

LPSTR WINAPI AnsiNext(LPCSTR lpchCurrentChar);
LPSTR WINAPI AnsiPrev(LPCSTR lpchStartChar, LPCSTR lpchCurrentChar);

410.2 Description

The AnsiNext() and AnsiPrev() functions move to the next or previous characters in the string respectively. AnsiPrev() requires a pointer to the starting character for reference.

410.3 Returns

These function return a pointer to the next or previous character in the string. AnsiNext() returns a pointer to the NULL character if it is encountered. AnsiPrev() returns a pointer to the starting character, if the lpchCurrentChar parameter is equal to the lpchStartChar parameter.

410.4 Errors

None.

410.5 Cross-References

AnsiLower(), AnsiUpper()

411 IsCharAlpha

411.1 Synopsis

BOOL WINAPI IsCharAlpha(char chTest);

411.2 Description

The IsCharAlpha() function tests if the character is in the set of alphabetic characters.

411.3 Returns

The function returns TRUE if the character is in the set. Otherwise, it returns FALSE.

411.4 Errors

None.

411.5 Cross-References

IsCharAlphaNumeric(), IsCharLower(), IsCharUpper

412 IsCharAlphaNumeric

412.1 Synopsis

BOOL WINAPI IsCharAlphaNumeric(char chTest);

412.2 Description

The IsCharAlphaNumeric() function tests if the character is in the set of alphabetic or numeric characters.
412.3 **Returns**  
This function returns TRUE if the character is in the set. Otherwise, it returns FALSE.

412.4 **Errors**  
None.

412.5 **Cross-References**  
*IsCharAlpha(), IsCharLower(), IsCharUpper*

---

413 **IsCharLower**

413.1 **Synopsis**  
BOOL WINAPI IsCharLower(char chTest);

413.2 **Description**  
The *IsCharLower()* function tests if the character is lower case.

413.3 **Returns**  
This function returns TRUE if the character is lower case. Otherwise, it returns FALSE.

413.4 **Errors**  
None.

413.5 **Cross-References**  
*IsCharUpper()*

---

414 **IsCharUpper**

414.1 **Synopsis**  
BOOL WINAPI IsCharUpper(chTest);

414.2 **Description**  
The *IsCharUpper()* function tests if the character is upper case.

414.3 **Returns**  
This function returns TRUE if the character is upper case. Otherwise, it returns FALSE.

414.4 **Errors**  
None.

414.5 **Cross-References**  
*IsCharLower()*

---

415 **lstrcmp, lstrcmpi**

415.1 **Synopsis**  
int WINAPI lstrcmp(LPCSTR lpszStr1, LPCSTR lpszStr2);
int WINAPI lstrcmpi(LPCSTR lpszStr1, LPCSTR lpszStr1);

415.2 **Description**  
The *lstrcmp()* and *lstrcmpi()* functions compare two strings. The *lstrcmp()* function is case sensitive, while the *lstrcmpi()* function is not.

415.3 **Returns**  
These functions return a value less than zero if *lpszStr1* is less than *lpszStr2*. It returns zero if the strings are equal, and greater than zero if *lpszStr1* is greater than *lpszStr2*. 
415.4 **Errors**  
None.

415.5 **Cross-References**  
lstrcpy()

416 **lstrcat, lstrcpyp, lstrcpyn**

416.1 **Synopsis**  
LPSTR WINAPI lstrcat(LPSTR lpszDest, LPCSTR lpszSrc);

LPSTR WINAPI lstrcpy(LPSTR lpszDest, LPCSTR lpszSrc);

LPSTR WINAPI lstrcpyn(LPSTR lpszDest, LPCSTR lpszSrc, int cChars);

416.2 **Description**  
The lstrcat() function concatenates the string lpszSrc to the end of lpszDest. The lstrcpy() and lstrcpyn() functions copy the contents from the string lpszSrc to the string lpszDest, including the NULL character. The lstrcpyn() function only copies cChars of string lpszSrc to lpszDest. It pads the string with NULL characters to the end of string lpszSrc.

416.3 **Returns**  
These functions return a pointer to lpszDest, if they are successful. Otherwise, they return NULL.

416.4 **Errors**  
None.

416.5 **Cross-References**  
lstrcmp()

417 **lstrlen**

417.1 **Synopsis**  
int WINAPI lstrlen(LPCSTR lpszString);

417.2 **Description**  
The lstrlen() function determines the length of the string.

417.3 **Returns**  
The lstrlen() function returns the number of characters contained in the string, not including the NULL terminator.

417.4 **Errors**  
None.

417.5 **Cross-References**  
lstrcmp()

418 **wsprintf, wvsprintf**

418.1 **Synopsis**  
int CDECL wsprintf(LPSTR lpszOut, LPCSTR lpszFmt, ...);

int WINAPI wvsprintf(LPSTR lpszOut, LPCSTR lpszFmt, const void * lpParams);

418.2 **Description**  
The wsprintf() and wvsprintf() functions format and convert the characters and values into the string lpszOut. The lpszFmt string contains the objects that control the conversion.
The `wvsprintf()` function is equivalent to the `wsprintf()` function except that the variable argument list is replaced by an array of values, specifying the arguments for the format string.

The format string has two types of objects, normal characters and conversion specifications. Normal characters are copied directly to the output string. A conversion specification begins with the character `%` and ends with a conversion character. The conversion process is performed on the next consecutive argument in the argument list. If the character following the `%` is not a valid format character, the single character is output to the string `lpszOut`.

Between the `%` and the conversion character, there may be one of the following:

- This object pads the output with blanks or zeros to left justify the output; if omitted, the output is right justified.
- This object pads the output with zeros to fill the field width.
- This object prefaces hexadecimal values with 0x for lowercase, or 0X for uppercase.
- This object is the minimum field width; the converted argument will be printed at least this wide or wider.
- This object is the minimum number of digits to be converted; if there are few digits, then the output is padded on the left with zeros.
- For strings, this object is the maximum number of characters to be converted.
- This object formats the argument as a character, a string or a number as shown below.

The following are valid conversion types:

- This type inserts a signed decimal integer argument.
- This type inserts a long signed decimal integer argument.
- This type inserts a long unsigned hexadecimal integer argument in lower case or upper case.
- This type inserts an unsigned integer argument.
- This type inserts a single character after conversion to an unsigned character.
- This type inserts characters from the string until a NULL is reached or characters indicated by the precision have been output.
- No argument is output; this type outputs a `%` character.

### 418.3 Returns

`wsprintf()` and `wvsprintf()` return the number of characters contained in the string `lpszOut`, not including the NULL terminator.

### 418.4 Errors

None.

### 418.5 Cross-References

`lstrcpy()`

---

### 419 IsDBCSLeadByte

#### 419.1 Synopsis

```c
BOOL IsDBCSLeadByte(BYTE TestChar);
```

#### 419.2 Description

The `IsDBCSLeadByte()` function identifies whether the character specified by the `TestChar` parameter is a lead byte, meaning it is the first character in a double-byte character set (DBCS).

The `TestChar` parameter identifies the character that needs to be tested. The current language driver determines whether the character is in the set. However, if no language driver is set then an internal function is used by the system. It should be pointed out here, that each double-byte character in a character set has unique lead bytes. The lead byte by themselves do not have any value, but the lead byte and the following byte, called a trailing byte, together represent a single character.
419.3 Returns
The function IsDBCSLeadByte() returns TRUE if the character is indeed a DBCS lead byte. Otherwise, it returns FALSE.

419.4 Errors
None.

419.5 Cross-References
GetKeyboardType()

420 ToAscii
420.1 Synopsis
int ToAscii(UINT VirtKeyCode, UINT ScanCode, BYTE * lpKeyStateBuff, DWORD * lpTransKeyBuff, UINT FlagState);

420.2 Description
The ToAscii() function converts the specified virtual-keycode and keyboard state to the corresponding windows character or characters. The VirtKeyCode parameter identifies the virtual-keycode to be converted. The ScanCode parameter identifies the hardware scan code of the key to be converted. If the key is not in the pressed state, the high-order bit of this value is set. The lpKeyStateBuff parameter contains a pointer to a 256-byte array which contains the current keyboard state. Each element of the array contains the state of one key, with the high-order byte indicating whether the key is in the pressed state. The lpTransKeyBuff parameter contains a pointer to the doubleword buffer, which will hold the translated system character or characters. The FlagState parameter identifies if the menu is active. If this value is set to 1, the menu is active. It is set to zero, if inactive.

ToAscii() does the conversion based on the virtual-key code, but in some cases the ScanCode parameter can be used to differentiate between a key in the pressed state and the released state. The scan-code is used in converting the ALT+number key combinations. Where a previous dead key is stored in the keyboard buffer, the parameters to the function ToAscii() may not be sufficient to convert the given virtual-key code.

420.3 Returns
If the function returns a negative value, the specified key is a dead key. Otherwise, the return value can have one of the following values and meaning.

2  Two characters were copied to the buffer; this is usually an accent and a dead-key character, when the dead key cannot be translated.
1  One system character was copied to the buffer.
0  The specified virtual key has no translation for the current state of the keyboard.

420.4 Errors
None.

420.5 Cross-References
OemKeyScan(), VkKeyScan()

421 AnsiToOem, AnsiToOemBuff
421.1 Synopsis
void AnsiToOem(const char _huge *WindowsSet, char _huge *OemSet);
void AnsiToOemBuff(LPCSTR WindowsSet, LPSTR OemSet, UINT BufferSize);

421.2 Description
The AnsiToOem() function takes the string defined by WindowsSet and converts it into the OEM format specified. The resultant string is stored in the buffer pointed to by OemSet.
The `AnsiToOemBuff()` function performs the same function as `AnsiToOem()`, but has the buffer size contained in the `BufferSize` parameter. `BufferSize` defaults to 64K, if it is given the value zero.

### 421.3 Returns
None.

### 421.4 Errors
None.

### 421.5 Cross-References
`OemToAnsi(), OemToAnsiBuff()`

---

### 422 OemToAnsi, OemToAnsiBuff

#### 422.1 Synopsis
```c
void OemToAnsi(const char *OemBuffer, char *WindowsBuffer);
void OemToAnsiBuff(LPCSTR OemBuffer, LPSTR WindowsBuffer, UINT BufferSize);
```

#### 422.2 Description
The `OemToAnsi()` function takes an OEM-defined string, `OemBuffer`, and converts it into a window string, placing the resultant string in the buffer, `WindowsBuffer`.

The `OemToAnsiBuff()` function performs the same function as `OemToAnsi()`, however, the size of `OemBuffer` is specified by `BufferSize`. `BufferSize` defaults to 64K, if it is given the value zero.

#### 422.3 Returns
None.

#### 422.4 Errors
None.

#### 422.5 Cross-References
`OemToAnsi(), OemToAnsiBuff()`

---

### 423 CopyRect, SetRect, SetRectEmpty, InflateRect, OffsetRect

#### 423.1 Synopsis
```c
typedef struct tagRECT {
    int left;
    int top;
    int right;
    int bottom;
} RECT, *LPRECT;

void CopyRect(LPRECT lprcDest, LPRECT lprcSrc);
void SetRect(LPRECT lprc, int nLeft, int nTop, int nRight, int nBottom);
void SetRectEmpty(LRECT lprc);
void InflateRect(LRECT lprc, int x, int y);
void OffsetRect(LPRECET lprc, int x, int y);
```

#### 423.2 Description
These functions modify the contents of the specified rectangle.
The `CopyRect()` function copies the elements from the source rectangle to the destination rectangle.

The `SetRect()` function copies the given parameters, `nLeft`, `nTop`, `nRight`, and `nBottom`, to the corresponding elements in the specified rectangle.

The `SetRectEmpty()` function sets each of the elements in the specified rectangle to zero.

The `InflateRect()` function adds `x` to the `right` and `left` elements, and `y` to the `top` and `bottom` elements of the specified triangle. Negative values of `x` or `y` shrink the rectangle in that dimension, while positive values increase the size of the rectangle in that direction.

The `OffsetRect()` function moves the specified rectangle by the amounts given. The `x` value is added to both the `left` and `right` element, while the `y` value is added to both the `top` and `bottom` elements of the given rectangle. Either of the `x` or `y` values can be negative to move the rectangle up or left, or positive to move the rectangle right or down.

423.3 Returns
None.

423.4 Errors
None.

423.5 Cross-References
`EqualRect()`, `IsRectEmpty()`, `PtInRect()`, `InflateRect()`, `OffsetRect()`, `IntersectRect()`, `UnionRect()`, `SubtractRect()`, `RECT`

424 EqualRect, IsRectEmpty, PtInRect

424.1 Synopsis
```c
typedef struct tagRECT {
    int left;
    int top;
    int right;
    int bottom;
} RECT, *LPRECT;

type struct tagPOINT {
    int x;
    int y;
} POINT, *LPPOINT;

BOOL EqualRect(LPRECT lprc1, LPRECT lprc2);
BOOL IsRectEmpty(LPRECT lprc);
BOOL PtInRect(LPRECT lprc, LPPOINT lppt);
```

424.2 Description
These functions test various conditions about a rectangle. `EqualRect()` compares each element of the first rectangle to its corresponding element in the second rectangle. If they are the same, the rectangles are equal. `IsRectEmpty()` checks to see if the given rectangle is empty. A rectangle is empty if either the `height` `(bottom - top)`, or `width` `(right - left)`, is less than or equal to zero.

`PtInRect()` checks to see if the point `lprc` lies within the rectangle.

424.3 Returns
`EqualRect()` returns TRUE if the two rectangles are equal. Otherwise, it returns FALSE. `IsRectEmpty()` returns TRUE if the rectangle is empty. Otherwise it returns FALSE. `PtInRect()` returns TRUE if the point is the rectangle, otherwise it returns FALSE.
PtInRect() returns TRUE if the point lppt is within the rectangle. It also returns TRUE if the point is on the top or left side. Otherwise, PtInRect() returns FALSE.

424.4 Errors
None.

424.5 Cross-References
CopyRect(), SetRect(), SetRectEmpty(), InflateRect(), OffsetRect(), IntersectRect(), UnionRect(), SubtractRect()

425 IntersectRect, UnionRect, SubtractRect
425.1 Synopsis
BOOL IntersectRect(LPRECT lprcDest, LPRECT lprcSrc, LPRECT lprcDiff);
BOOL UnionRect(LPRECT lprcDest, LPRECT lprcSrc, LPRECT lprcDiff);
BOOL SubtractRect(LPRECT lprcDest, LPRECT lprcSrc, LPRECT lprcDiff);

425.2 Description
These functions combine two source rectangles, lprcSrc and lprcDiff, to generate a new rectangle, which is stored in lprcDest.
IntersectRect() creates a new rectangle consisting of the largest rectangle that is contained in both source rectangles.
UnionRect() creates the minimum rectangle that completely encloses both of the two source rectangles.
SubtractRect() creates a new rectangle that is the result of subtracting one rectangle from another. The resulting rectangle is identical to the source rectangle if the subtraction rectangle does not completely contain the height or width of the source rectangle.

425.3 Returns
If the result of the operation creates an empty rectangle, the result is FALSE. If it is not empty, the result is TRUE.

425.4 Errors
None.

425.5 Cross-References
CopyRect(), SetRect(), SetRectEmpty(), InflateRect(), OffsetRect(), EqualRect(), IsRectEmpty(), PtInRect(), RECT

426 OutputDebugString
426.1 Synopsis
void OutputDebugString(LPCSTR lpszStr);

426.2 Description
The OutputDebugString() function outputs the null-terminated string lpszStr to the debugger. The debugger must be running for the output to appear.

426.3 Returns
None.

426.4 Errors
None.

426.5 Cross-References
DebugOutput()
427 DebugOutput

427.1 Synopsis
void _cdecl DebugOutput(UINT flags, LPCSTR lpszFmt, ...);

427.2 Description
The DebugOutput() function outputs a message to the debugger. The debugger must be running for the output to appear. The flags parameter controls the type of message the debugger receives and is one of the following:

- DBF_TRACE This value reports that no error has occurred.
- DBF_WARNING This value reports a warning that may or may not be an error.
- DBF_ERROR This value reports an error resulting from an API function call.
- DBF_FATAL This value reports an error that will terminate the application.

The application formats the output in the same manner as wprintf(). The lpszFmt string contains the objects that control the conversion. See the description for wprintf() for detailed formatting information.

The ... argument is for zero or more arguments, the number and type of which are determined by the format string lpszFmt.

427.3 Returns
None.

427.4 Errors
None.

427.5 Cross-References
OutputDebugString()

428 FatalAppExit

428.1 Synopsis
void FatalAppExit(UINT action, LPCSTR lpszMessage);

428.2 Description
The FatalAppExit() function displays the null-terminated string lpszMessage in a message box. The message is displayed on a single line, so it should not be longer than 35 characters. When the user acknowledges the message, the application is terminated.

The action parameter is reserved and must be zero.

428.3 Returns
None.

428.4 Errors
None.

428.5 Cross-References
FatalExit()

429 FatalExit

429.1 Synopsis
void FatalExit(int nErrCode);
429.2 **Description**
The `FatalExit()` function displays the error code `nErrCode` in the debugger and halts execution. If the debugger is running, the user can terminate the application or continue. If the debugger is not running, the application is terminated.

429.3 **Returns**
None.

429.4 **Errors**
None.

429.5 **Cross-References**
`FatalAppExit()`

---

430 **QuerySendMessage**

430.1 **Synopsis**
`BOOL QuerySendMessage(HANDLE hOne, HANDLE hTwo, HANDLE hThree, LPMSG lpMsg);`

430.2 **Description**
The `QuerySendMessage()` function determines whether a message sent by the `SendMessage()` function was originally sent by the current task. If the message is being sent along with other tasks, the `QuerySendMessage()` function puts it into the MSG structure, specified by the `lpMsg` parameter. Parameters `hOne`, `hTwo` and `hThree` must be NULL.

430.3 **Returns**
The `QuerySendMessage()` function returns FALSE if the message originates within the current task. Otherwise, it returns TRUE.

430.4 **Errors**
None.

430.5 **Cross-References**
`SendMessage(), PostMessage(), ReplyMessage()`

---

431 **LockInput**

431.1 **Synopsis**
`BOOL LockInput(HANDLE hOne, HWND hwndInput, BOOL fLock);`

431.2 **Description**
If the `fLock` parameter is TRUE, the `LockInput()` function locks keyboard and mouse input to all tasks except the current one. The locked window becomes system modal, that is it receives all input events. If the `fLock` parameter is FALSE, all locked windows are unlocked. The `hOne` parameter should be NULL.

431.3 **Returns**
The `LockInput()` function returns TRUE if it is successful. Otherwise, it returns FALSE.

431.4 **Errors**
None.

431.5 **Cross-References**
`Yield(), DirectedYield()`
**432 FlashWindow**

**432.1 Synopsis**

BOOL FlashWindow(HWND hWnd, BOOL bInvert);

**432.2 Description**

The *FlashWindow()* function flashes a window by toggling its title bar. This toggle effect is the same as if the window was activated and deactivated, or vice versa.

The *bInvert* parameter specifies to flash the window or restore it to its original state. If *bInvert* is TRUE, the window is flashed from one state to another. If it is FALSE, the window is restored to its original state.

If the window is minimized, the *bInvert* flag is ignored and its icon is flashed.

**432.3 Returns**

The function returns TRUE if the window was active before the call, and FALSE if it was inactive.

**432.4 Errors**

None.

**432.5 Cross-References**

*MessageBeep()*

---

**433 MessageBeep**

**433.1 Synopsis**

void MessageBeep(UINT uAlert);

**433.2 Description**

The *MessageBeep()* function plays a sound corresponding to the alert level specified by *uAlert*. The sound played at each alert level is determined by the entry in the [sounds] section of the WIN.INI file.

The alert level *uAlert* can be one of the following. The entry specified is located in the [sounds] section of the WIN.INI file.

-1 Standard beep.

MB_ICONASTERISK Sound in the SystemAsterisk entry.
MB_ICONEXCLAMATION Sound in the SystemExclamation entry.
MB_ICONHAND Sound in the SystemHand entry.
MB_ICONQUESTION Sound in the SystemQuestion entry.
MB_OK Sound in the SystemDefault entry.

**433.3 Returns**

None.

**433.4 Errors**

None.

**433.5 Cross-References**

*MessageBox()*

---

**434 MessageBox**

**434.1 Synopsis**

int MessageBox(HWND hWndParent, LPCSTR lpszMessage, LPCSTR lpszTitle, UINT uStyle);
434.2 Description
The MessageBox() function displays the null-terminated string lpszMessage in a dialog box window. The dialog box title is set to the null-terminated string lpszTitle. The hWndParent parameter is the parent of the dialog box, this parameter may be set to NULL for no parent. The uStyle parameter allows control over the contents and behavior of the dialog box. It can be a combination of the following values:

- MB_ABORTRETRYIGNORE: The dialog has Abort, Retry, and Ignore push buttons.
- MB_OK: The dialog only contains the OK push button.
- MB_OKCANCEL: The dialog has OK and Cancel push buttons.
- MB_RETRYCANCEL: The dialog has Retry and Cancel push buttons.
- MB_YESNO: The dialog has Yes and No push buttons.
- MB_YENOCANCEL: The dialog has Yes, No and Cancel push buttons.
- MB_DEFButtons1: The first button will be the default; this is the default case if no other buttons are specified as default.
- MB_DEFButtons2: The second button is the default.
- MB_DEFButtons3: The third button is the default.
- MB_ICON INFORMATION: The information icon appears in the dialog box.
- MB_ICONASTERISK: This value is the same as the MB_ICON INFORMATION option.
- MB_ICONEXCLAMATION: The exclamation or caution icon appears in the dialog box.
- MB_ICONHAND: The stop icon appears in the dialog box.
- MB_ICONSTOP: This value is the same as the MB_ICON HAND.
- MB_ICONQUESTION: The question icon appears in the dialog box.
- MB_APPLMODAL: The user must respond to the dialog before any of the current application windows can be accessed; the windows of separate applications may be accessed.
- MB_SYSTEMMODAL: All applications are suspended until the user responds to the dialog; the user cannot access any other windows.
- MB_TASKMODAL: This value is the same as MB_APPLMODAL, except that if hWndParent is NULL, all top-level windows are disabled.

The default handling of the dialog is MB_APPLMODAL if neither the MB_SYSTEMMODAL nor the MB_TASKMODAL options are used.

434.3 Returns
This function returns zero, if the dialog box fails to display. If successful, the return value is one of the following:

- IDABORT: The Abort button was selected.
- IDCANCEL: The Cancel button was selected.
- IDIGNORE: The Ignore button was selected.
- IDNO: The No button was selected.
- IDOK: The OK button was selected.
- IDRETRY: The Retry button was selected.
- IDYES: The Yes button was selected.

If the dialog has a Cancel button and the Esc key is pressed, the dialog returns IDCANCEL.
434.4 Errors
None.

434.5 Cross-References
MessageBeep()

---

435 SetErrorMode

435.1 Synopsis
UINT SetErrorMode(UINT fuErrorMode);

435.2 Description
The SetErrorMode() function allows the application to control the appearance of MS-DOS interrupt error messages. The fuErrorMode parameter can be a combination of the following values:

- **SEM_FAILCRITICALERRORS**: Do not display the critical-error-handler message box and return the error to the calling application.
- **SEM_NOGPFAULTERROBOX**: Do not display the general-protection-fault message box.
- **SEM_NOOPENFILEERRORBOX**: Do not display a message box when the system fails to find a file.

435.3 Returns
The SetErrorMode() function returns the previous value of error-mode flag, if it is successful.

435.4 Errors
None.

435.5 Cross-References
None.

---

436 GetExpandedName

436.1 Synopsis
int GetExpandedName(LPCSTR SourceFile, LPSTR OriginalName);

436.2 Description
The GetExpandedName() function is used to return the name of the original compressed file, SourceFile. The extracted filename is placed in OriginalName. The prerequisites of using this function are:

- The file be compressed with COMPRESS.EXE.
- The file be compressed with the /t option.

If SourceFile is not compressed, OriginalName is extracted from SourceFile.

436.3 Returns
If GetExpandedName() is completed successfully, then TRUE is returned.

If GetExpandedName() is unsuccessful, an error code is returned (a value less than zero). One of the more common error messages to be returned is LZERROR_BADINHANDLE, which means that the SourceFile is an incorrect file handle. This can happen in many situations, most likely, not having used the /t option in compressing the file.

436.4 Errors
None.

436.5 Cross-References
None.
ChooseColor

Synopsis

typedef struct tagCHOOSECOLOR {
    DWORD lStructSize;
    HWND hWndOwner;
    HINSTANCE hInstance;
    COLORREF rgbResult;
    COLORREF *lpCustColors;
    DWORD Flags;
    LPARAM lCustData;
    UINT (CALLBACK *lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR lpTemplateName;
} CHOOSECOLOR, *LPCHOOSECOLOR;

BOOL ChooseColor(LPCHOOSECOLOR lpcc);

Description

The ChooseColor() function provides the user with a modal dialog box, under the control of the lpcc parameter, to allow for the interactive selection of a color or colors. The operation of the dialog box is controlled by the Flags member of the CHOOSECOLOR structure. Constant values for the Flags member are the following:

CC_ENABLEHOOK
CC_ENABLETEMPLATE
CC_ENABLETEMPLATEHANDLE
CC_FULLOPEN
CC_PREVENTFULLOPEN
CC_RGBINIT
CC_SHOWHELP

The layout of the dialog box controls are defined by a built-in CHOOSECOLOR dialog box template or by values passed into the ChooseColor() function.

The default layout consists of a simple array of colors for the user to select, while the expanded layout allows the user to define and select customized colors. The expanded view can be selected by specifying CC_FULLOPEN, and can be disabled by setting the CC_PREVENTFULLOPEN flag.

Alternative dialog box control layouts can be specified by setting either the CC_ENABLETEMPLATE or CC_ENABLETEMPLATEHANDLE flags. The CC_ENABLETEMPLATE selects a user defined dialog box template resource that is accessed by using the values of the hInstance and the lpTemplateName members. If CC_ENABLETEMPLATEHANDLE is specified, the value of hInstance is a handle to a block of memory defining the in-memory instance of the dialog box template.

If the CC_ENABLEHOOK flag is set, the hook function pointed to by the lpfnHook member is called for any message that is processed by the ChooseColor() function. If the hook function processes the message, then the function should return TRUE, to prevent the ChooseColor() dialog box procedure from further processing the message. The lCustData parameter is used to pass data through the ChooseColor() function to the user defined hook function.

If the CC_SHOWHELP flag is set, the dialog box procedure adds a HELP button that can be pressed by the user to receive user defined help. If hWndOwner is specified, it denotes the window that owns the dialog box, and receives any help messages generated during the operation of ChooseColor() when the user presses HELP.
The `rgbResult` and `lpCustColor` members are set on input and define the initial values to be selected when the dialog box is initialized, provided the CC_RGBINIT flag is set. On output, these values contain the selected color values, if the function is successful.

437.3 Returns
The function returns TRUE if it is successful. The function returns FALSE if it is aborted by the user, or if an error is encountered. If the function is successful, `ChooseColor()` updates the `rgbResult` with the users selected color. If custom colors are defined, the `lpCustColors` array is filled out with 16 customized colors defined by the user.

437.4 Errors
If the `ChooseColor()` function is unsuccessful, or encounters a failure, a common dialog box error value is set. This error value can be retrieved by using the `CommDlgExtendedError()` function. The defined errors area:

- **CDERR-INITIALIZATION**: ChooseColor() encountered an error during the dialog box initialization, such as not enough memory, unable to create a control, or missing components specified by Flags.
- **CDERR_FINDRESFAILURE**: ChooseColor() was unable to find one of the required resource templates.
- **CDERR_LOADRESFAILURE**: ChooseColor() was unable to load the required dialog box template.
- **CDERR_LOCKRESFAILURE**: ChooseColor() was unable to lock the dialog box template resource needed to build the dialog box.
- **CDERR_LOADSTRFAILURE**: One of the required string resources was unable to be loaded.
- **CDERR_NOHINSTANCE**: Flags required a valid hInstance member to be specified.
- **CDERR_NOHOOK**: Flags required a hook function to be specified.
- **CDERR_NOTEMPLATE**: Flags required a valid template to be specified.
- **CDERR_STRUCTSIZE**: The size specified for the `CHOOSECOLOR` structure was incorrect.

437.5 Cross-References
`CommDlgExtendedError()`, `CHOOSECOLOR`

438 ChooseFont

438.1 Synopsis

```c
typedef struct tagCHOOSEFONT {
    DWORD     lStructSize;
    HWND      hwndOwner;
    HDC       hdc;
    LPLOGFONT lpLogFont;
    int       iPointSize;
    DWORD     Flags;
    COLORREF  rgbColors;
    LPARAM    lCustData;
    UINT      (CALLBACK *lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    HINSTANCE hInstance;
    LPSTR     lpszStyle;
    UINT      nFontType;
    int       nSizeMin;
} CHOOSEFONT;
```
int nSizeMax;

} CHOOSEFONT, *LPCHOOSEFONT;

BOOL ChooseFont(LPCHOOSEFONT lpcf);

438.2 Description

The ChooseFont() function provides the user with a modal dialog box, under the control of the lpcf parameter, which allows for the interactive selection of a font. The dialog box allows all the aspects of a font to be modified. This includes the size, as well as Typeface and special effects such as bold, italics, underline, strikethrough, and color. The operation of the dialog box is controlled by the Flags member of the CHOOSEFONT structure. The layout of the dialog box controls are defined by a built-in CHOOSEFONT dialog box template or by values passed to the ChooseFont() function.

Alternative dialog box control layouts can be specified by setting either the CF_ENABLETEMPLATE or CF_ENABLETEMPLATEHANDLE flags. The CF_ENABLETEMPLATE selects a user defined dialog box template resource that is accessed by using the values of the hInstance and the lpTemplateName members. If CF_ENABLETEMPLATEHANDLE is specified, the hInstance value is a handle to a block of memory defining the in-memory instance of the dialog box template. If the CF_ENABLEHOOK flag is set, then a hook function is called (by the lpfnHook member) for any message that is processed by the ChooseFont() dialog box procedure. If the hook function processes the message, then it should return a TRUE value to prevent the ChooseFont() dialog box procedure from further processing the message. The lCustData member is used to pass data through the ChooseFont() function to the user-defined hook function. On the WM_INITDIALOG message, a pointer to the CHOOSEFONT structure is passed in LPARAM. From the pointer location, the lCustData member is available.

If the CF_SHOWHELP flag is set, the dialog box procedure adds a HELP button that can be pressed to receive user-defined help. If hWndOwner is specified, it denotes the window that owns the dialog box and receives any help messages that are generated during the operation of ChooseFont(), when the user presses HELP.

The lpLogFont and rgbColors members are set on input to define the initial values selected when the dialog box is initialized, if the CF_INITTOLOGFONTSTRUCT and CF_EFFECTS flags are set. If the function runs successfully, the following happens; on output, the structure specified by the lpLogFont member is updated with the selected logical font and the rgbColors value is updated to the color chosen by the user. If the CF_USESTYLE flag is set and the dialog box procedure is successful, then the lpszStyle member describes the initial style to use and it sets the style that is selected by the user. The following flags are used to control the types of fonts dealt with during the execution of ChooseFont():

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF_FIXEDPITCHONLY</td>
<td>Allow only fixed pitch fonts to be displayed.</td>
</tr>
<tr>
<td>CF_FORCEFONTEXIST</td>
<td>Make sure that the user selected font actually exists.</td>
</tr>
<tr>
<td>CF_LIMITSIZE</td>
<td>Use the nSizeMin and nSizeMax fields to limit the users selection to fonts in that range.</td>
</tr>
<tr>
<td>CF_PRINTERFONTS</td>
<td>Allow only fonts that are supported by the currently selected printer identified by the hdc parameter.</td>
</tr>
<tr>
<td>CF_SCALABLEONLY</td>
<td>Allow only scaleable fonts to be selected.</td>
</tr>
<tr>
<td>CF_WYSIWYG</td>
<td>Allow only the selection of those fonts that can be displayed on the screen and the printer.</td>
</tr>
<tr>
<td>CF_BOTH</td>
<td>Allow both printer and screen fonts to be displayed.</td>
</tr>
<tr>
<td>CF_ANSIONLY</td>
<td>Allow only fonts that are compatible with the ANSI character set.</td>
</tr>
</tbody>
</table>

If the CF_APPLY flag is set, the ChooseFont() dialog box procedure enables the APPLY button. If the user presses this button, the selected font, text colors, and special effects are applied to the hdc member and are returned to the user in the appropriate fields of the CHOOSEFONT structure.

438.3 Returns

The function returns TRUE if the function is successful. The function returns FALSE if the function is aborted by the user or if an error is encountered. If the function is successful, ChooseFont() updates the structure specified by
the lpLogFont member with the selected logical font information. If requested by the user and enabled by the caller, the function updates the hdc member with the desired font and selected font color.

438.4 Errors
If the ChooseFont() function is unsuccessful, it returns FALSE. The error code from the CommDlgExtendedError() function returns one of the following:

CDERR_DIALOGFAILURE The dialog box cannot be created.
CDERR_INITIALIZATION A common dialog function encountered an error during initialization of the dialog box; for example, not enough memory, unable to create a control.
CDERR_FINDRESFAILURE A common dialog function is unable to find one of the required resource templates.
CDERR_LOADRESFAILURE The dialog box procedure is unable to load the required dialog box template.
CDERR_LOCKRESFAILURE The dialog box procedure is unable to lock the dialog box template resource needed to build the dialog box (CDERR_LOADSTRFAILURE). One of the string resources required is unable to be loaded.
CDERR_NOHINSTANCE Flags required the specification of a valid hInstance member.
CDERR_NOHOOK Flags required the specification of a hook function.
CDERR_NOTEMPLATE Flags required the specification of a valid template.
CDERR_REGISTERMSGFAIL The function RegisterWindowMessage() failed to register the defined help string message.
CDERR_STRUCTSIZE The size specified for the CHOOSEFONT structure is incorrect.
CFERR_NOFONTS No fonts are found that match the user request.
CFERR_MAXLESSTHANMIN The maximum size of the font specified is less than the minimum sized specified.

438.5 Cross-References
CommDlgExtendedError(), CHOOSEFONT

439 FindText, ReplaceText
439.1 Synopsis
typedef struct tagFINDREPLACE{
439.2 Description

The `FindText()` and `ReplaceText()` functions create modeless dialog boxes, under the control of the `lpfr` parameter, that make it possible for users to find text within a document.

Alternative dialog box control layouts can be specified either by setting the `FR_ENABLETEMPLATE` or `FR_ENABLETEMPLATEHANDLE` flag. The `FR_ENABLETEMPLATE` flag selects a user defined dialog box template resource that is accessed by using the values of the `hInstance` and the `lpTemplateName` members. If `FR_ENABLETEMPLATEHANDLE` is specified, the `hInstance` value is a handle to a block of memory defining the in-memory instance of the dialog box template. If the `FR_ENABLEHOOK` flag is set, then the `lpfnHook` function is called for any message that will be processed by the dialog box procedure. If the hook function processes the message, then it should return a non-zero value to prevent the dialog box procedure from further processing the message. The `lCustData` member is available to pass data through the dialog box function to the user-defined hook function. On the WM_INITDIALOG message, a pointer to the `FINDREPLACE` structure is passed in LPARAM. From the pointer location, the `lCustData` member is available.

If the `FR_SHOWHELP` flag is set, the dialog box procedure adds a HELP button that can be pressed to receive user defined help. If the `hwndOwner` member is specified, it denotes the window that owns the dialog box, and receives any help messages that are generated while operating the dialog box.

The following flags can be set to further configure the dialog box layout:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR_HIDEMATCHCASE</td>
<td>This flag causes the Match Case checkbox to be disabled by hiding the control, thus preventing the user from changing its value.</td>
</tr>
<tr>
<td>FR_NOMATCHCASE</td>
<td>This flag causes the Match Case checkbox to be disabled.</td>
</tr>
<tr>
<td>FR_HIDEWHOLEWORLD</td>
<td>This flag causes the Whole Word checkbox to be disabled by hiding the control, thus preventing the user from changing its value.</td>
</tr>
<tr>
<td>FR_NOWHOLEWORLD</td>
<td>This flag causes the Whole Word check box to be disabled.</td>
</tr>
<tr>
<td>FR_HIDEUPDOWN</td>
<td>This flag causes the Up Down buttons to be disabled by hiding the control, thus preventing the user from changing its value.</td>
</tr>
<tr>
<td>FR_NOUPDOWN</td>
<td>This flag causes the Up Down buttons to be disabled.</td>
</tr>
</tbody>
</table>

After the dialog box is created, it communicates with its parent window through the use of the special registered messages, `FINDMSGSTRING` and `REPLACEMSGSTRING`. The dialog box procedure fills out the `lpszFindWhat` and `lpszReplaceWith` buffers and updates the `Flags` member to reflect the current dialog box values before sending the message to `hwndOwner`. The LPARAM of this message is a pointer to the `FINDREPLACE` structure where the `Flags` value has been modified to contain the following bits:
FR_FINDNEXT  The application should search for the next occurrence of the string specified by the lpstrFindWhat member; the search should use the additional flag bits to determine what direction to search, whether to match upper and lower case, and whether a wholeword should be matched.

FR_REPLACE  The application should replace the current selection string given by lpstrFindWhat with the string given by lpstrReplaceWith.

FR_REPLACEALL  Similar to FR_REPLACE, this replaces all occurrences of the string given by lpstrFindWhat with lpstrReplaceWith.

FR_DOWN  The search should proceed downward in the document.

FR_MATCHCASE  The match for a string should be identical to the string given by lpstrFindWhat.

FR_WHOLEWORD  The match for a string should be identical to a whole word only and not parts of a word.

439.3  Returns
FindText() and ReplaceText() return the handle of the system modeless dialog box, or NULL, if there is an error in creating the dialog box.

439.4  Errors
If the FindText() or ReplaceText() function is unsuccessful, it returns NULL. The error code can be retrieved by calling the CommDlgExtendedError() function. The value returned by the function is one of the following error codes:

CDERR_DIALOGFAILURE  The dialog box could not be created.
CDERR_INITIALIZATION  A common dialog function encountered an error during initialization of the dialog box, such as not enough memory, unable to create a control.
CDERR_FINDRESFAILURE  A common dialog function was unable to find one of the resource templates that are required to function.
CDERR_LOADRESFAILURE  The dialog box procedure was unable to load the required dialog box template.
CDERR_LOCKRESFAILURE  The dialog box procedure was unable to lock the dialog box template resource needed to build the dialog box.
CDERR_LOADSTRFAILURE  One of the string resources required was unable to be loaded.
CDERR_NOINSTANCE  Flags required a valid hInstance member to be specified.
CDERR_NOHOOK  Flags required a hook function to be specified.
CDERR_NOTEMPLATE  Flags required a valid template to be specified.
CDERR_REGISTERMSGFAIL  The function RegisterWindowMessage() failed to register the defined help string message.
CDERR_STRUCTSIZE  The size specified for the FINDREPLACE structure was incorrect.

439.5  Cross-References
IsDialogMessage(), RegisterWindowMessage(), CommDlgExtendedError(), FINDREPLACE

440  GetOpenFileName, GetSaveFileName

440.1  Synopsis
typedef struct tagOPENFILENAME{
    DWORD lStructSize;
} OPENFILENAME;
HWND hwndOwner;
HINSTANCE hInstance;
LPCSTR lpstrFilter;
LPSTR lpstrCustomFilter;
DWORD nMaxCustFilter;
DWORD nFilterIndex;
LPSTR lpstrFile;
DWORD nMakeFile
LPSTR lpstrFileTitle;
DWORD nMaxFileTitle;
LPCSTR lpstrInitialDir;
LPCSTR lpstrTitle;
DWORD Flags;
UINT nFileOffset;
UINT nFileExtension;
LPCSTR lpstrDefExt;
LPARAM lCustData;
UINT (CALLBACK *lpfnHook)(HWND, UINT, WPARAM, LPARAM);
LPCSTR lpTemplateName;

} OPENFILENAME, *LPOPENFILENAME;

BOOL GetOpenFileName(LPOPENFILENAME lpof);
BOOL GetSaveFileName(LPOPENFILENAME lpof);

440.2 Description
The GetOpenFileName() and GetSaveFileName() functions provide the user with a modal dialog box, under the control of the lpof parameter, which allows for the interactive selection of a file, with the ability to open, create and verify the file. The operation of the dialog box is controlled by the Flags member of the OPENFILENAME structure. The layout of the dialog box controls are defined by the built-in GETOPENFILENAME and GETSAVEFILENAME dialog box template or by values passed in the LPOPENFILENAME structure.

Alternative dialog box control layouts can be specified by setting either the OFN_ENABLETEMPLATE or OFN_ENABLETEMPLATEHANDLE flags. The OFN_ENABLETEMPLATE selects a user defined dialog box template resource that is accessed by using the values of the hInstance and the lpTemplateName members. If OFN_ENABLETEMPLATEHANDLE is specified, the hInstance value is a handle to a block of memory defining the in-memory instance of the dialog box template. If the OFN_ENABLEHOOK flag is set, then the lpfnHook function is called for any message that is processed by the dialog box procedure. If the hook function processes the message, it should return a non-zero value to prevent the dialog box procedure from further processing the message. The lCustData member is available to pass data through the dialog box function to the user defined hook function.

A pointer to the OPENFILENAME structure is passed in lParam of the WM_INITDIALOG message. From there the lCustData member is available.

If the OFN_SHOWHELP flag is set, the dialog box procedure adds a HELP button that is pressed by the user to receive user-defined help. If hWndOwner is specified, it denotes the window that owns the dialog box, and receives any help messages generated during the operation of modal dialog box procedure, when the user presses HELP.
If the OFN_HIDEREADONLY flag is set, then the Read Only check box is hidden during dialog box initialization. If the OFN_READONLY flag is set, then the Read Only check box is initialized and displayed. When the dialog box procedure completes successfully, this bit will contain the last state of the Read Only check box.

The dialog box procedures use the following flags to control the operation of the file dialog boxes:

- **OFN_FILEMUSTEXIST**: Only files listed in the file list box may be entered by the user in the filename edit control; filenames that do not match bring up a message box indicating that only matching names are allowed.
- **OFN_PATHMUSTEXIST**: Similar to the OFN_FILEMUSTEXIST flag, the user may enter valid pathnames in the filename edit control.
- **OFN_NOCHANGEDIR**: On exiting from the dialog box procedure, the dialog box restores the current working directory to its first initialized state.
- **OFN_NOREADONLYRETURN**: This ensures that the selected filename cannot be read-only or in a read-only directory.
- **OFN_NOTESTFILECREATE**: For the `GetSaveFileName()` function, this flag prevents the function from creating the file specified by the user.
- **OFN_NOVALIDATE**: If a hook procedure is used, the filename selected by the user is validated by filling out the `OPENFILENAME` structure and sending the register message FILEOKSTRING; this flag prevents the dialog box from attempting to validate the filename.
- **OFN_OVERWRITEPROMPT**: This flag causes the `GetSaveFileName()` function to prompt the user if an attempt is made to select an existing file.

### 440.3 Returns

The function returns TRUE if it is successful, or if the OK button is pressed to exit the dialog, or a filename is selected with a double-click. The function returns FALSE if the function is aborted by the user or if an error is encountered. If the function is successful, the following fields are updated by the dialog box procedure:

- **nFilterIndex**: This field represents the index of the filters that were last active.
- **lpstrFile**: This field is filled out with the complete pathname of the desired filename; its size is limited by the `nMaxFile` member.
- **lpstrFileTitle**: This field represents just the filename and any extension, with no path information; it either contains the filename and any extension or it is NULL. Its size is limited in length by the `nMaxFileTitle` member.
- **Flags**: The OFN_EXTENSIONDIFFERENT and OFN_READONLY flags are updated to reflect the current settings.
- **nFileOffset**: This field is set to the index in `lpstrFile` that starts the actual filename; this field excludes all path information.
- **nFileExtension**: This field is set to the index in `lpstrFile` that starts the actual extension of the filename; this field excludes all path information.

### 440.4 Errors

If `GetOpenFileName()` or `GetSaveFileName()` are unsuccessful, they return NULL. The error code is determined from `CommDlgExtendedError()`, which returns one of the following:

- **CDERR_DIALOGFAILURE**: The dialog box cannot be created.
- **CDERR_INITIALIZATION**: A common dialog function encountered an error during initialization of the dialog box, such as not enough memory or unable to create a control.
- **CDERR_FINDRESFAILURE**: A common dialog function is unable to find one of the required resource templates.
CDERR_LOADRESFAILURE The dialog box procedure is unable to load the required dialog box template.
CDERR_LOCKRESFAILURE The dialog box procedure is unable to lock the dialog box template resource needed to build the dialog box.
CDERR_LOADSTRFAILURE One of the string resources required cannot be loaded.
CDERR_NOHINSTANCE Flags required a valid hInstance member to be specified.
CDERR_NOHOOK Flags required a hook function to be specified.
CDERR_NOTEMPLATE Flags required a valid template to be specified.
CDERR_REGISTERMSGFAIL The function RegisterWindowMessage() failed to register the defined help string message.
CDERR_STRUCTSIZE The size specified for the OPENFILENAME structure is incorrect.
FNERR_INVALIDFILENAME The filename is not a legal filename.

440.5 Cross-References
RegisterWindowMessage(), CommDlgExtendedError(), OPENFILENAME

441 GetFileTitle
441.1 Synopsis
int GetFileTitle(LPCSTR lpszFile, LPSTR lpszTitle, UINT nSize);

441.2 Description
The GetFileTitle() function is a utility that extracts the actual filename from a filename specification, lpszFile, that includes path information. The filename specification must be a valid filename or an error occurs. To be valid the function must be non-null and contain no wildcard characters. It also must not be a directory reference and it must fit into the file title buffer. The actual filename is stored in the buffer lpszTitle. The nSize parameter is the size of lpszTitle in bytes.

441.3 Returns
GetFileTitle() returns zero if successful. If the filename supplied is not a valid filename, a negative number is returned. If the buffer is too small, a positive number is returned that identifies the required size of the file title buffer including a null terminator.

441.4 Errors
None.

441.5 Cross-References
None.

442 PrintDlg
442.1 Synopsis
BOOL PrintDlg(PRINTDLG *PrintDlgPtr);

442.2 Description
The PrintDlg() function shows the Print or Print Setup common dialog box. The PrintDlgPtr parameter is a pointer to a PRINTDLG structure that contains initialization information for the dialog box.

442.3 Returns
If the PrintDlg() function configures the printer, it returns TRUE. If the user closes the dialog box by pressing the Cancel button or by selecting the System menu’s Close menu item, the PrintDlg() function returns FALSE. If the following sequence of steps are performed, the PrintDlg() function will also return FALSE:
1) The user presses the Setup button.
2) The user presses the OK button in the Print Setup dialog box.
3) The user presses the Cancel button in the Print dialog box.

The function \texttt{CommDlgExtendedError()} can be used to retrieve an error value.

442.4 Errors
None.

442.5 Cross-References
PRINTDLG

443 \texttt{CommDlgExtendedError}

443.1 Synopsis
\texttt{DWORD CommDlgExtendedError(void);}

443.2 Description
The last error encountered during execution of one of the common dialog functions is saved and can be retrieved by
this function. Executing any common dialog box procedure successfully will clear the saved value.

443.3 Returns
If the last common dialog function was successful, the \texttt{CommDlgExtendedError()} function returns zero. Otherwise,
the \texttt{CommDlgExtendedError()} function returns one of the following:

\begin{itemize}
  \item CDERR_DIALOGFAILURE \quad The dialog box could not be created.
  \item CDERR_INITIALIZATION \quad A common dialog function encountered an error during initialization of
  \quad the dialog box, such as not enough memory, or unable to create a control.
  \item CDERR_FINDRESFAILURE \quad A common dialog function was unable to find one of the resource
  \quad templates that are required to function.
  \item CDERR_LOADRESFAILURE \quad The dialog box procedure was unable to load the required dialog box
  \quad template.
  \item CDERR_LOCKRESFAILURE \quad The dialog box procedure was unable to lock the dialog box template
  \quad resource needed to build the dialog box.
  \item CDERR_LOADSTRFAILURE \quad One of the string resources required was unable to be loaded.
  \item CDERR_NOHINSTANCE \quad \texttt{Flags} required a valid hInstance parameter to be specified.
  \item CDERR_NOHOOK \quad \texttt{Flags} required a hook function to be specified.
  \item CDERR_NOTEMPLATE \quad \texttt{Flags} required a valid template to be specified.
  \item CDERR_REGISTERMSGFAIL \quad The function \texttt{RegisterWindowMessage()} failed to register the defined
  \quad help string message.
  \item CDERR_STRUCTSIZE \quad The size specified for the structure was incorrect.
\end{itemize}

443.4 Errors
None.

443.5 Cross-References
\texttt{ChooseColor()}, \texttt{ChooseFont()}, \texttt{FindText()}, \texttt{ReplaceText()}, \texttt{GetFileTitle()}, \texttt{GetOpenFileName()}, \texttt{GetSaveFileName()}, \texttt{PrintDlg()}


444  MulDiv

444.1  Synopsis
int MulDiv(int Multiplicand, int Multiplier, int Divisor);

444.2  Description
The MulDiv() function performs the following operation:

\[(\text{Multiplicand} \times \text{Multiplier}) / \text{Divisor} = \text{return value}\]

444.3  Returns
This function returns the result of the multiplication and division. If either an overflow occurs or the divisor is zero.
(the system is trying to divide by zero) the return value will be -32,768.

444.4  Errors
None.

444.5  Cross-References
None.
Printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland
Fax: +41 22 849.60.01
Internet: helpdesk@ecma.ch

Files can be downloaded from our FTP site, ftp.ecma.ch, logging in as anonymous and giving your E-mail address as password. This Standard is available from library ECMA-ST as MSWord 6.0 files (E-234-V1.DOC, E-234-V2.DOC, E-234-V3.DOC), as PostScript files (E-234-V1.PSC, E-234-V2.PSC, E-234-V3.PSC) and as Acrobat files (E-234-V1.PDF, E-234-V2.PDF, E-234-V3.PDF).

The ECMA site can be reached also via a modem. The phone number is +41 22 735.33.29, modem settings are 8/n/1. Telnet (at ftp.ecma.ch) can also be used.

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

This Standard ECMA-234 is available free of charge in printed form and as a file.
See inside cover page for instructions