+Sync's Win32 Reference 1.0

+Cracker
+HCU Graduate

I put this together for my own use, but think that others might use it as well. I feel it is much more handy than the entire reference, although it can by no means replace it because it is not complete. However, as far as functions I need to look up in the interest of cracking this help file has almost all of them. I get about 96% of the functions I need in only 120kb of disk space. If there is a function that I have left out (I could not find hmemcpy for example) please let me know, I will include it. Also I would like to include a 'cracking notes' section with each function. If you would like to write such a section for any of the functions please do so.

API Functions

sync1@iname.com

Win32 API Functions

RegSaveKey
RegSetValue
RegSetValueEx
SetDlgItemInt
SetDlgItemText
SetFileAttributes
SetTimer
SetWindowText
ToAscii
ToAsciiEx
WriteFile
WriteFileEx
WritePrivateProfileString

GetCurrentDirectory

The GetCurrentDirectory function retrieves the current directory for the current process.

```
DWORD GetCurrentDirectory(
DWORD nBufferLength,     // size, in characters, of directory buffer
LPTSTR lpBuffer          // address of buffer for current directory
);
```

Parameters

nBufferLength

Specifies the length, in characters, of the buffer for the current directory string. The buffer length must include room for a terminating null character.

lpBuffer

Points to the buffer for the current directory string. This null-terminated string specifies the absolute path to the current directory.

Return Values

If the function succeeds, the return value specifies the number of characters written to the buffer, not including the terminating null character.

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

If the buffer pointed to by lpBuffer is not large enough, the return value specifies the required size of the buffer, including the number of bytes necessary for a terminating null character.

See Also

CreateDirectory, GetSystemDirectory, GetWindowsDirectory, RemoveDirectory, SetCurrentDirectory

GetDlgItem

The GetDlgItem function retrieves the handle of a control in the specified dialog box.

```
HWND GetDlgItem(
HWND hDlg,        // handle of dialog box
int nIDDlgItem    //identifier of control
);
```

Parameters

hDlg

Identifies the dialog box that contains the control.

nIDDlgItem

Specifies the identifier of the control to be retrieved.

Return Values

If the function succeeds, the return value is the window handle of the given control.

If the function fails, the return value is NULL, indicating an invalid dialog box handle or a nonexistent control.

Remarks

You can use the GetDlgItem function with any parent-child window pair, not just with dialog boxes. As long as the hDlg parameter specifies a parent window and the child window has a unique identifier (as specified by the hMenu parameter in the CreateWindow or CreateWindowEx function that created the child window), GetDlgItem returns a valid handle to the child window.

See Also

CreateWindow, CreateWindowEx, GetDlgItemInt, GetDlgItemText

GetDlgItemInt

The GetDlgItemInt function translates the text of a specified control in a dialog box into an integer value.

```
UINT GetDlgItemInt(
HWND hDlg,              // handle to dialog box
int nIDDlgItem,         // control identifier
BOOL *lpTranslated,   // points to variable to receive success/failure indicator
BOOL bSigned            // specifies whether value is signed or unsigned
);
```

Parameters

hDlg

Handle to the dialog box that contains the control of interest.

nIDDlgItem

Dialog item identifier that specifies the control whose text is to be translated.

lpTranslated

Points to a Boolean variable that receives a function success/failure value. TRUE indicates success, FALSE indicates failure.

This parameter is optional: it can be NULL. In that case, the function returns no information about success or failure.

bSigned

Specifies whether the function should examine the text for a minus sign at the beginning and return a signed integer value if it finds one. TRUE specifies that this should be done, FALSE that it should not.

Return Values

If the function succeeds, the variable pointed to by lpTranslated is set to TRUE, and the return value is the translated value of the control text.

If the function fails, the variable pointed to by lpTranslated is set to FALSE, and the return value is zero. Note that, since zero is a possible translated value, a return value of zero does not by itself indicate failure.

If lpTranslated is NULL, the function returns no information about success or failure.

If the bSigned parameter is TRUE, specifying that the value to be retrieved is a signed integer value, cast the return value to an int type.

Remarks

The GetDlgItemInt function retrieves the text of the given control by sending the control a WM_GETTEXT message. The function translates the retrieved text by stripping any extra spaces at the beginning of the text and then converting the decimal digits. The function stops translating when it reaches the end of the text or encounters a nonnumeric character.

If the bSigned parameter is TRUE, the GetDlgItemInt function checks for a minus sign (-) at the beginning of the text and translates the text into a signed integer value. Otherwise, the function creates an unsigned integer value.

The GetDlgItemInt function returns zero if the translated value is greater than INT_MAX (for signed numbers) or UINT_MAX (for unsigned numbers).

See Also

GetDlgItem

GetDlgItemText

The GetDlgItemText function retrieves the title or text associated with a control in a dialog box.

```
UINT GetDlgItemText(
HWND hDlg,              // handle of dialog box
int nIDDlgItem,         // identifier of control
LPTSTR lpString, // address of buffer for text
int nMaxCount           // maximum size of string
);
```

Parameters

hDlg

Identifies the dialog box that contains the control.

nIDDlgItem

Specifies the identifier of the control whose title or text is to be retrieved.

lpString

Points to the buffer to receive the title or text.

nMaxCount

Specifies the maximum length, in characters, of the string to be copied to the buffer pointed to by lpString. If the length of the string exceeds the limit, the string is truncated.

Return Values

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero.

Remarks

The GetDlgItemText function sends a WM_GETTEXT message to the control.

See Also

GetDlgItemInt

GetFileTime

The GetFileTime function retrieves the date and time that a file was created, last accessed, and last modified.

```
BOOL GetFileTime(
HANDLE hFile,                          // identifies the file
LPFILETIME lpCreationTime,       // address of creation time
LPFILETIME lpLastAccessTime, // address of last access time
LPFILETIME lpLastWriteTime      // address of last write time
);
```

Parameters

hFile

Identifies the files for which to get dates and times. The file handle must have been created with GENERIC_READ access to the file.

lpCreationTime

Points to a FILETIME structure to receive the date and time the file was created. This parameter can be NULL if the application does not require this information.

lpLastAccessTime

Points to a FILETIME structure to receive the date and time the file was last accessed. The last access time includes the last time the file was written to, read from, or, in the case of executable files, run. This parameter can be NULL if the application does not require this information.

lpLastWriteTime

Points to a FILETIME structure to receive the date and time the file was last written to. This parameter can be NULL if the application does not require this information.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The FAT and New Technology file systems support the file creation, last access, and last write time values.

Windows 95: The precision of the time for a file in a FAT file system is 2 seconds. The time precision for files in other file systems, such as those connected through a network depends on the file system but may also be limited by the remote device.

See Also

GetSystemTime

GetLocalTime

The GetLocalTime function retrieves the current local date and time.

VOID GetLocalTime(
LPSYSTEMTIME lpSystemTime // address of system time structure
);

Parameters

lpSystemTime

Points to a SYSTEMTIME structure to receive the current local date and time.

Return Values

This function does not return a value.

See Also

GetSystemTime

GetOpenFileName

The GetOpenFileName function creates an Open common dialog box that lets the user specify the drive, directory, and the name of a file or set of files to open.

BOOL GetOpenFileName(
LPOPENFILENAME lpofn   // address of structure with initialization data
);

Parameters

lpofn

Pointer to an OPENFILENAME structure that contains information used to initialize the dialog box. When GetOpenFileName returns, this structure contains information about the user's file selection.

Return Values

If the user specifies a filename and clicks the OK button, the return value is nonzero. The buffer pointed to by the lpstrFile member of the OPENFILENAME structure contains the full path and filename specified by the user.

If the user cancels or closes the Open dialog box or an error occurs, the return value is zero. To get extended error information, call the CommDlgExtendedError function, which can return one of the following values:

CDERR_FINDRESFAILURE
CDERR_NOHINSTANCE
CDERR_INITIALIZATION
CDERR_NOHOOK
CDERR_LOCKRESFAILURE
CDERR_NOTEMPLATE
CDERR_LOADRESFAILURE
CDERR_STRUCTSIZE
CDERR_LOADSTRFAILURE
FNERR_BUFFERTOOSMALL
CDERR_MEMALLOCFAILURE
FNERR_INVALIDFILENAME
CDERR_MEMLOCKFAILURE
FNERR_SUBCLASSFAILURE

Remarks

By default, Windows 95 and Windows NT version 4.0 display a new version of the Open dialog box that provides user-interface features that are similar to the Windows Explorer. You can provide an OFNHookProc hook procedure for an Explorer-style Open dialog box. To enable the hook procedure, set the OFN_EXPLORER and OFN_ENABLEHOOK flags in the Flags member of the OPENFILENAME structure and specify the address of the hook procedure in the lpfnHook member.

Windows 95 and Windows NT 4.0 continue to support the old-style Open dialog box for applications that want to maintain a user-interface consistent with the Windows 3.1 or Windows NT 3.51 user-interface. To display the old-style Open dialog box, enable an OFNHookProcOldStyle hook procedure and ensure that the OFN_EXPLORER flag is not set.

GetPrivateProfileInt

The GetPrivateProfileInt function retrieves an integer associated with a key in the specified section of the given initialization file. This function is provided for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
UINT GetPrivateProfileInt(
LPCTSTR lpAppName, // address of section name
LPCTSTR lpKeyName, // address of key name
INT nDefault,                   // return value if key name is not found
LPCTSTR lpFileName    // address of initialization filename
);
```

Parameters

lpAppName

Points to a null-terminated string containing the section name in the initialization file.

lpKeyName

Points to the null-terminated string containing the key name whose value is to be retrieved. This value is in the form of a string; the GetPrivateProfileInt function converts the string into an integer and returns the integer.

nDefault

Specifies the default value to return if the key name cannot be found in the initialization file.

lpFileName

Points to a null-terminated string that names the initialization file. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory.

Return Values

If the function succeeds, the return value is the integer equivalent of the string following the specified key name in the specified initialization file. If the key is not found, the return value is the specified default value. If the value of the key is less than zero, the return value is zero.

Remarks

The function searches the file for a key that matches the name specified by the lpKeyName parameter under the section name specified by the lpAppName parameter. A section in the initialization file must have the following form:

```
[section]
key=value
.
.
.
```

The GetPrivateProfileInt function is not case-sensitive; the strings in lpAppName and lpKeyName can be a combination of uppercase and lowercase letters.

An application can use the GetProfileInt function to retrieve an integer value from the WIN.INI file.

Windows NT:

Calls to private profile functions may be mapped to the registry instead of to the specified initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following keys:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as CONTROL.INI, SYSTEM.INI, and WINFILE.INI. In these cases, the GetPrivateProfileInt function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The Win32 Profile functions (Get/WriteProfile*, Get/WritePrivateProfile*) use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say myfile.ini, under IniFileMapping:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping\myfile.ini

2. Look for the section name specified by lpAppName. This will be a named value under myfile.ini, or a subkey of myfile.ini, or will not exist.

3. If the section name specified by lpAppName is a named value under myfile.ini, then that value specifies where in the registry you will find the keys for the section.

4. If the section name specified by lpAppName is a subkey of myfile.ini, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as "<No Name>") that specifies the default location in the registry where you will find the key.

5. If the section name specified by lpAppName does not exist as a named value or as a subkey under myfile.ini, then there will be an unnamed value (shown as "<No Name>") under myfile.ini that specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey for myfile.ini, or if there is no entry for the section name, then look for the actual myfile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the ini file mapping:

! - this character forces all writes to go both to the registry and to the .INI file on disk.

# - this character causes the registry value to be set to the value in the Windows 3.1 .INI file when a new user logs in for the first time after setup.

@ - this character prevents any reads from going to the .INI file on disk if the requested data is not found in the registry.

USR: - this prefix stands for HKEY_CURRENT_USER, and the text after the prefix is relative to that key.

SYS: - this prefix stands for HKEY_LOCAL_MACHINE\SOFTWARE, and the text after the prefix is relative to that key.

See Also

GetProfileInt

GetPrivateProfileSection

The GetPrivateProfileSection function retrieves all of the keys and values for the specified section from an initialization file. This function is provided for compatibility with 16-bit applications written for Windows. Win32-based applications should store initialization information in the registry.

Windows 95:

The specified profile section must not exceed 32K.

Windows NT:

The specified profile section has no size limit.

```
DWORD GetPrivateProfileSection(
LPCTSTR lpAppName,          // address of section name

LPTSTR lpReturnedString, // address of return buffer
DWORD nSize,                          // size of return buffer
LPCTSTR lpFileName             // address of initialization filename
);
```

Parameters

lpAppName

Points to a null-terminated string containing the section name in the initialization file.

lpReturnedString

Points to a buffer that receives the key name and value pairs associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

nSize

Specifies the size, in characters, of the buffer pointed to by the lpReturnedString parameter.

Windows 95:

The maximum buffer size is 32,767 characters.

Windows NT:

There is no maximum buffer size.

lpFileName

Points to a null-terminated string that names the initialization file. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory.

Return Values

The return value specifies the number of characters copied to the buffer, not including the terminating null character. If the buffer is not large enough to contain all the key name and value pairs associated with the named section, the return value is equal to nSize minus two.

Remarks

The data in the buffer pointed to by the lpReturnedString parameter consists of one or more null-terminated strings, followed by a final null character. Each string has the following format:

key=string

The GetPrivateProfileSection function is not case-sensitive; the string pointed to by the lpAppName parameter can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the specified initialization file are allowed while the key name and value pairs for the section are being copied to the buffer pointed to by the lpReturnedString parameter.

Windows NT:

Calls to private profile functions may be mapped to the registry instead of to the specified initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following keys:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as CONTROL.INI, SYSTEM.INI, and WINFILE.INI. In these cases, the GetPrivateProfileSection function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The Win32 Profile functions (Get/WriteProfile*, Get/WritePrivateProfile*) use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say myfile.ini, under IniFileMapping:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping\myfile.ini

2. Look for the section name specified by lpAppName. This will be a named value under myfile.ini, or a subkey of myfile.ini, or will not exist.

3. If the section name specified by lpAppName is a named value under myfile.ini, then that value specifies where in the registry you will find the keys for the section.

4. If the section name specified by lpAppName is a subkey of myfile.ini, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as "<No Name>") that specifies the default location in the registry where you will find the key.

5. If the section name specified by lpAppName does not exist as a named value or as a subkey under myfile.ini, then there will be an unnamed value (shown as "<No Name>") under myfile.ini that specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey for myfile.ini, or if there is no entry for the section name, then look for the actual myfile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the ini file mapping:

! - this character forces all writes to go both to the registry and to the .INI file on disk.

# - this character causes the registry value to be set to the value in the Windows 3.1 .INI file when a new user logs in for the first time after setup.

@ - this character prevents any reads from going to the .INI file on disk if the requested data is not found in the registry.

USR: - this prefix stands for HKEY_CURRENT_USER, and the text after the prefix is relative to that key.

SYS: - this prefix stands for HKEY_LOCAL_MACHINE\SOFTWARE, and the text after the prefix is relative to that key.

See Also

GetProfileInt

GetPrivateProfileString

The GetPrivateProfileString function retrieves a string from the specified section in an initialization file. This function is provided for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
DWORD GetPrivateProfileString(
LPCTSTR lpAppName, // points to section name
LPCTSTR lpKeyName, // points to key name
LPCTSTR lpDefault,        // points to default string
LPTSTR lpReturnedString, // points to destination buffer
DWORD nSize, // size of destination buffer
LPCTSTR lpFileName // points to initialization filename
);
```

Parameters

lpAppName

Points to a null-terminated string that specifies the section containing the key name. If this parameter is NULL, the GetPrivateProfileString function copies all section names in the file to the supplied buffer.

lpKeyName

Pointer to the null-terminated string containing the key name whose associated string is to be retrieved. If this parameter is NULL, all key names in the section specified by the lpAppName parameter are copied to the buffer specified by the lpReturnedString parameter.

lpDefault

Pointer to a null-terminated default string. If the lpKeyName key cannot be found in the initialization file, GetPrivateProfileString copies the default string to the lpReturnedString buffer. This parameter cannot be NULL.

Avoid specifying a default string with trailing blank characters. The function inserts a null character in the lpReturnedString buffer to strip any trailing blanks.

Windows 95: Although lpDefault is declared as a constant parameter, Windows 95 strips any trailing blanks by inserting a null character into the lpDefault string before copying it to the lpReturnedString buffer.

Windows NT: Windows NT does not modify the lpDefault string. This means that if the default string contains trailing blanks, the lpReturnedString and lpDefault strings will not match when compared using the lstrcmp function.

lpReturnedString

Pointer to the buffer that receives the retrieved string.

nSize

Specifies the size, in characters, of the buffer pointed to by the lpReturnedString parameter.

lpFileName

Pointer to a null-terminated string that names the initialization file. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory.

Return Values

If the function succeeds, the return value is the number of characters copied to the buffer, not including the terminating null character.

If neither lpAppName nor lpKeyName is NULL and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a null character, and the return value is equal to nSize minus one.

If either lpAppName or lpKeyName is NULL and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two null characters. In this case, the return value is equal to nSize minus two.

Remarks

The GetPrivateProfileString function searches the specified initialization file for a key that matches the name specified by the lpKeyName parameter under the section heading specified by the lpAppName parameter. If it finds the key, the function copies the corresponding string to the buffer. If the key does not exist, the function copies the default character string specified by the lpDefault parameter. A section in the initialization file must have the following form:

```
[section]
key=string
.
.
.
```

If lpAppName is NULL, GetPrivateProfileString copies all section names in the specified file to the supplied buffer. If lpKeyName is NULL, the function copies all key names in the specified section to the supplied buffer. An application can use this method to enumerate all of the sections and keys in a file. In either case, each string is followed by a null character and the final string is followed by a second null character. If the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two null characters.

If the string associated with lpKeyName is enclosed in single or double quotation marks, the marks are discarded when the GetPrivateProfileString function retrieves the string.

The GetPrivateProfileString function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

To retrieve a string from the WIN.INI file, use the GetProfileString function.

Windows NT:

Calls to private profile functions may be mapped to the registry instead of to the specified initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following keys:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as CONTROL.INI, SYSTEM.INI, and WINFILE.INI. In these cases, the GetPrivateProfileString function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The Win32 Profile functions (Get/WriteProfile*, Get/WritePrivateProfile*) use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say myfile.ini, under IniFileMapping:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping\myfile.ini

2. Look for the section name specified by lpAppName. This will be a named value under myfile.ini, or a subkey of myfile.ini, or will not exist.

3. If the section name specified by lpAppName is a named value under myfile.ini, then that value specifies where in the registry you will find the keys for the section.

4. If the section name specified by lpAppName is a subkey of myfile.ini, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as "<No Name>") that specifies the default location in the registry where you will find the key.

5. If the section name specified by lpAppName does not exist as a named value or as a subkey under myfile.ini, then there will be an unnamed value (shown as "<No Name>") under myfile.ini that specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey for myfile.ini, or if there is no entry for the section name, then look for the actual myfile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the ini file mapping:

! - this character forces all writes to go both to the registry and to the .INI file on disk.

# - this character causes the registry value to be set to the value in the Windows 3.1 .INI file when a new user logs in for the first time after setup.

@ - this character prevents any reads from going to the .INI file on disk if the requested data is not found in the registry.

USR: - this prefix stands for HKEY_CURRENT_USER, and the text after the prefix is relative to that key.

SYS: - this prefix stands for HKEY_LOCAL_MACHINE\SOFTWARE, and the text after the prefix is relative to that key.

See Also

GetProfileSint

GetSystemTime

The GetSystemTime function retrieves the current system date and time. The system time is expressed in Coordinated Universal Time (UTC).

VOID GetSystemTime(
LPSYSTEMTIME lpSystemTime   // address of system time structure
);

Parameters

lpSystemTime

Points to a SYSTEMTIME structure to receive the current system date and time.

Return Values

This function does not return a value.

See Also

GetLocalTime, GetSystemTimeAdjustment, SetSystemTime, SYSTEMTIME

GetSystemTimeAsFileTime

The GetSystemTimeAsFileTime function obtains the current system date and time. The information is in Coordinated Universal Time (UTC) format.

VOID GetSystemTimeAsFileTime(
LPFILETIME lpSystemTimeAsFileTime // pointer to a file time structure
);

Parameters

lpSystemTimeAsFileTime

Pointer to a FILETIME structure to receive the current system date and time in UTC format.

Return Values

This function does not return a value.

Remarks

The GetSystemTimeAsFileTime function is equivalent to the following code sequence:

```
FILETIME ft;
SYSTEMTIME st;

GetSystemTime(&st);
SystemTimeToFileTime(&st,&ft);
```

See Also

FILETIME, GetSystemTime, SYSTEMTIME, SystemTimeToFileTime

GetTickCount

The GetTickCount function retrieves the number of milliseconds that have elapsed since Windows was started.

DWORD GetTickCount(VOID)

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the number of milliseconds that have elapsed since Windows was started.

Remarks

The elapsed time is stored as a DWORD value. Therefore, the time will wrap around to zero if Windows is run continuously for 49.7 days.

Windows NT: To obtain the time elapsed since the computer was started, look up the System Up Time counter in the performance data in the registry key HKEY_PERFORMANCE_DATA. The value returned is an 8 byte value.

GetWindowLong

The GetWindowLong function retrieves information about the specified window. The function also retrieves the 32-bit (long) value at the specified offset into the extra window memory of a window.

```
LONG GetWindowLong(
HWND hWnd, // handle of window
int nIndex    // offset of value to retrieve
);
```

Parameters

hWnd

Identifies the window and, indirectly, the class to which the window belongs.

nIndex

Specifies the zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus four; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value, specify one of the following values:

GWL_EXSTYLE
Retrieves the extended window styles.

GWL_STYLE
Retrieves the window styles.

GWL_WNDPROC
Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

GWL_HINSTANCE
Retrieves the handle of the application instance.

GWL_HWNDPARENT
Retrieves the handle of the parent window, if any.

GWL_ID
Retrieves the identifier of the window.

GWL_USERDATA
Retrieves the 32-bit value associated with the window. Each window has a corresponding 32-bit value intended for use by the application that created the window.

The following values are also available when the hWnd parameter identifies a dialog box:

DWL_DLGPROC
Retrieves the address of the dialog box procedure, or a handle representing the address of the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.

DWL_MSGRESULT
Retrieves the return value of a message processed in the dialog box procedure.

DWL_USER
Retrieves extra information private to the application, such as handles or pointers.

Return Values

If the function succeeds, the return value is the requested 32-bit value.

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

Remarks

Reserve extra window memory by specifying a nonzero value in the cbWndExtra member of the WNDCLASS structure used with the RegisterClass function.

See Also

CallWindowProc, GetWindowWord, RegisterClass, SetParent

GetWindowText

The GetWindowText function copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied.

int GetWindowText(
HWND hWnd, // handle of window or control with text
LPTSTR lpString, // address of buffer for text
int nMaxCount // maximum number of characters to copy
);

Parameters

hWnd

Identifies the window or control containing the text.

lpString

Points to the buffer that will receive the text.

nMaxCount

Specifies the maximum number of characters to copy to the buffer, including the NULL character. If the text exceeds this limit, it is truncated.

Return Values

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character. If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero. To get extended error information, call GetLastError.

This function cannot retrieve the text of an edit control in another application.

Remarks

This function causes a WM_GETTEXT message to be sent to the specified window or control.

This function cannot retrieve the text of an edit control in another application.

See Also

~~GetWindowTextLength~~

GetWindowTextLength

The GetWindowTextLength function retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). If the specified window is a control, the function retrieves the length of the text within the control.

int GetWindowTextLength(
HWND hWnd // handle of window or control with text
);

Parameters

hWnd

Identifies the window or control.

Return Values

If the function succeeds, the return value is the length, in characters, of the text. Under certain conditions, this value may actually be greater than the length of the text. For more information, see the following Remarks section.

If the window has no text, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function causes a WM_GETTEXTLENGTH message to be sent to the specified window or control.

Under certain conditions, the GetWindowTextLength function may return a value that is larger than the actual length of the text. This occurs with certain mixtures of ANSI and Unicode, and is due to the operating system allowing for the possible existence of DBCS characters within the text. The return value, however, will always be at least as large as the actual length of the text; you can thus always use it to guide buffer allocation. This behavior can occur when an application uses both ANSI functions and common dialogs, which use Unicode. It can also occur when an application uses the ANSI flavor of GetWindowTextLength with a window whose window procedure is Unicode, or the Unicode flavor with a window whose window procedure is ANSI.

To obtain the exact length of the text, use the WM_GETTEXT, LB_GETTEXT, or CB_GETLBTEXT messages, or the GetWindowText function.

See Also

CB_GETLBTEXT, GetWindowText, LB_GETTEXT, SetWindowText

GetWindowWord

The GetWindowWord function retrieves a 16-bit (word) value at the specified offset into the extra window memory for the specified window.

WORD GetWindowWord(
HWND hWnd, // handle of window
int nIndex // offset of value to retrieve
);

Parameters

hWnd

Identifies the window and, indirectly, the class to which the window belongs.

nIndex

Specifies the zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus two; for example, if you specified 10 or more bytes of extra window memory, a value of 8 would be an index to the fifth 16-bit integer.

Return Values

If the function succeeds, the return value is the requested 16-bit value.

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

Remarks

Reserve extra window memory by specifying a nonzero value in the cbWndExtra member of the WNDCLASS structure used with the RegisterClass function.

The GWW_ values are obsolete in Win32. You must use the GetWindowLong function to retrieve information about the window.

See Also

GetParent, GetWindowLong, RegisterClass, SetParent, SetWindowLong

Beep

The Beep function generates simple tones on the speaker. The function is synchronous; it does not return control to its caller until the sound finishes.

```
BOOL Beep(
DWORD dwFreq, // sound frequency, in hertz
DWORD dwDuration // sound duration, in milliseconds
);
```

Parameters

dwFreq

Windows NT:

Specifies the frequency, in hertz, of the sound. This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).

Windows 95:

The parameter is ignored.

dwDuration

Windows NT:

Specifies the duration, in milliseconds, of the sound.

Windows 95:

The parameter is ignored.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Windows 95:

The Beep function ignores the dwFreq and dwDuration parameters. On computers with a sound card, the function plays the default sound event. On computers without a sound card, the function plays the standard system beep.

See Also

MessageBeep

CharUpper

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

```
LPTSTR CharUpper(
LPTSTR lpsz   // single character or pointer to string
);
```

Parameters

lpsz

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

Return Values

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

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

There is no indication of success or failure. Failure is rare.

Remarks

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

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

See Also

CharLower

CharLower

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

LPTSTR CharLower(
LPTSTR lpsz // single character or pointer to string
);

Parameters

lpsz

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

Return Values

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

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

There is no indication of success or failure. Failure is rare.

Remarks

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

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

See Also

CharLowerBuff, CharUpper

CompareString

The CompareString function compares two character strings, using the locale specified by the given identifier as the basis for the comparison.

int CompareString(
LCID Locale, // locale identifier
DWORD dwCmpFlags, // comparison-style options
LPCTSTR lpString1, // pointer to first string
int cchCount1, // size, in bytes or characters, of first string
LPCTSTR lpString2, // pointer to second string
int cchCount2 // size, in bytes or characters, of second string
);

Parameters

Locale

Specifies the locale used for the comparison. This parameter can be one of the following predefined locale identifiers:

LOCALE_SYSTEM_DEFAULT
The system's default locale.

LOCALE_USER_DEFAULT
The current user's default locale.

This parameter can also be a locale identifier created by the MAKELCID macro.

dwCmpFlags

A set of flags that indicate how the function compares the two strings. By default, these flags are not set. This parameter can specify zero to get the default behavior, or it can be any combination of the following values:

NORM_IGNORECASE
Ignore case.

NORM_IGNOREKANATYPE
Do not differentiate between Hiragana and Katakana characters. Corresponding Hiragana and Katakana characters compare as equal.

NORM_IGNORENONSPACE
Ignore nonspacing characters.

NORM_IGNORESYMBOLS
Ignore symbols.

NORM_IGNOREWIDTH
Do not differentiate between a single-byte character and the same character as a double-byte character.

SORT_STRINGSORT
Treat punctuation the same as symbols.

lpString1

Points to the first string to be compared.

cchCount1

Specifies the size, in bytes (ANSI version) or characters (Unicode version), of the string pointed to by the lpString1 parameter. If this parameter is - 1, the string is assumed to be null terminated and the length is calculated automatically.

lpString2

Points to the second string to be compared.

cchCount2

Specifies the size, in bytes (ANSI version) or characters (Unicode version), of the string pointed to by the lpString2 parameter. If this parameter is - 1, the string is assumed to be null terminated and the length is calculated automatically.

Return Values

If the function succeeds, the return value is one of the following values:

CSTR_LESS_THAN
The string pointed to by the lpString1 parameter is less in lexical value than the string pointed to by the lpString2 parameter.

CSTR_EQUAL
The string pointed to by lpString1 is equal in lexical value to the string pointed to by lpString2.

CSTR_GREATER_THAN
The string pointed to by lpString1 is greater in lexical value than the string pointed to by lpString2.

If the function fails, the return value is zero. To get extended error information, call GetLastError. GetLastError may return one of the following error codes:

ERROR_INVALID_FLAGS

ERROR_INVALID_PARAMETER

Remarks

Notice that if the return value is 2, the two strings are "equal" in the collation sense, though not necessarily identical.

To maintain the C run-time convention of comparing strings, the value 2 can be subtracted from a nonzero return value. The meaning of < 0, ==0 and > 0 is then consistent with the C run times.

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, then the return value will indicate that the longer string is greater. For more information about locale identifiers, see Locale Identifiers.

Typically, strings are compared using what is called a "word sort" technique. In a word sort, all punctuation marks and other nonalphanumeric characters, except for the hyphen and the apostrophe, come before any alphanumeric character. The hyphen and the apostrophe are treated differently than the other nonalphanumeric symbols, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list.

If the SORT_STRINGSORT flag is specified, strings are compared using what is called a "string sort" technique. In a string sort, the hyphen and apostrophe are treated just like any other nonalphanumeric symbols: they come before the alphanumeric symbols.

The following table shows a list of words sorted both ways:

Word Sort	String Sort	Word Sort	String Sort
billet	bill's	t-ant	t-ant
bills	billet	tanya	t-aria
bill's	bills	t-aria	tanya
cannot	can't	sued	sue's
cant	cannot	sues	sued
can't	cant	sue's	sues
con	co-op	went	we're
coop	con	were	went
co-op	coop	we're	were

The lstrcmp and lstrcmpi functions use a word sort. The CompareString and LCMapString functions default to using a word sort, but use a string sort if their caller sets the SORT_STRINGSORT flag.

The CompareString function is optimized to run at the highest speed when dwCmpFlags is set to 0 or NORM_IGNORECASE, and cchCount1 and cchCount2 have the value -1.

The CompareString function ignores Arabic Kashidas during the comparison. Thus, if two strings are identical save for the presence of Kashidas, CompareString returns a value of 2; the strings are considered "equal" in the collation sense, though they are not necessarily identical.

See Also

FoldString, GetSystemDefaultLCID, GetUserDefaultLCID, LCMapString, lstrcmp

CreateDialog

The CreateDialog macro creates a modeless dialog box from a dialog box template resource. The CreateDialog macro uses the CreateDialogParam function.

```
HWND CreateDialog(
HINSTANCE hInstance, // handle to application instance
LPCTSTR lpTemplate, // identifies dialog box template name
HWND hWndParent, // handle to owner window
DLGPROC lpDialogFunc // pointer to dialog box procedure
);
```

Parameters

hInstance

Identifies an instance of the module whose executable file contains the dialog box template.

lpTemplate

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

hWndParent

Identifies the window that owns the dialog box.

lpDialogFunc

Points to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

Return Values

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

If the function fails, the return value is NULL.

Remarks

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

After CreateDialog returns, the application displays the dialog box (if it is not already displayed) by using the ShowWindow function. The application destroys the dialog box by using the DestroyWindow function.

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

See Also

CreateDialogIndirect, CreateDialogIndirectParam, CreateDialogParam, CreateWindowEx, DestroyWindow, DialogBox, DialogProc, ShowWindow, WM_INITDIALOG, WM_SETFONT

CreateFile

The CreateFile function creates or opens the following objects and returns a handle that can be used to access the object:

· files
· pipes
· mailslots
· communications resources
· disk devices (Windows NT only)
· consoles
· directories (open only)

HANDLE CreateFile(
LPCTSTR lpFileName, // pointer to name of the file
DWORD dwDesiredAccess, // access (read-write) mode
DWORD dwShareMode, // share mode
LPSECURITY_ATTRIBUTES lpSecurityAttributes, // pointer to security attributes
DWORD dwCreationDistribution, // how to create
DWORD dwFlagsAndAttributes, // file attributes
HANDLE hTemplateFile // handle to file with attributes to copy
);

Parameters

lpFileName

Points to a null-terminated string that specifies the name of the object (file, pipe, mailslot, communications resource, disk device, console, or directory) to create or open.

If *lpFileName is a path, there is a default string size limit of MAX_PATH characters. This limit is related to how the CreateFile function parses paths.

Windows NT: You can use paths longer than MAX_PATH characters by calling the wide (W) version of CreateFile and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing. This lets you use paths that are nearly 32,000 Unicode characters long. You must use fully-qualified paths with this technique. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\tom_1\hotstuff\coolapps" is seen as "\\tom_1\hotstuff\coolapps".

dwDesiredAccess

Specifies the type of access to the object. An application can obtain read access, write access, read-write access, or device query access. This parameter can be any combination of the following values.

0
Specifies device query access to the object. An application can query device attributes without accessing the device.

GENERIC_READ
Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with GENERIC_WRITE for read-write access.

GENERIC_WRITE
Specifies write access to the object. Data can be written to the file and the file pointer can be moved. Combine with

GENERIC_READ for read-write access.

dwShareMode

Set of bit flags that specifies how the object can be shared. If dwShareMode is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

To share the object, use a combination of one or more of the following values:

FILE_SHARE_DELETE
Windows NT only: Subsequent open operations on the object will succeed only if delete access is requested.

FILE_SHARE_READ
Subsequent open operations on the object will succeed only if read access is requested.

FILE_SHARE_WRITE
Subsequent open operations on the object will succeed only if write access is requested.

lpSecurityAttributes

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

Windows NT: The lpSecurityDescriptor member of the structure specifies a security descriptor for the object. If lpSecurityAttributes is NULL, the object gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect on files.

Windows 95: The lpSecurityDescriptor member of the structure is ignored.

dwCreationDistribution

Specifies which action to take on files that exist, and which action to take when files do not exist. For more information about this parameter, see the Remarks section. This parameter must be one of the following values:

CREATE_NEW
Creates a new file. The function fails if the specified file already exists.

CREATE_ALWAYS
Creates a new file. The function overwrites the file if it exists.

OPEN_EXISTING
Opens the file. The function fails if the file does not exist.
See the Remarks section for a discussion of why you should use the OPEN_EXISTING flag if you are using the CreateFile function for devices, including the console.

OPEN_ALWAYS
Opens the file, if it exists. If the file does not exist, the function creates the file as if dwCreationDistribution were CREATE_NEW.

TRUNCATE_EXISTING
Opens the file. Once opened, the file is truncated so that its size is zero bytes. The calling process must open the file with at least GENERIC_WRITE access. The function fails if the file does not exist.

dwFlagsAndAttributes

Specifies the file attributes and flags for the file.

Any combination of the following attributes is acceptable for the dwFlagsAndAttributes parameter, except all other file attributes override FILE_ATTRIBUTE_NORMAL.

FILE_ATTRIBUTE_ARCHIVE
The file should be archived. Applications use this attribute to mark files for backup or removal.

FILE_ATTRIBUTE_COMPRESSED
The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.

FILE_ATTRIBUTE_HIDDEN
The file is hidden. It is not to be included in an ordinary directory listing.

FILE_ATTRIBUTE_NORMAL
The file has no other attributes set. This attribute is valid only if used alone.

FILE_ATTRIBUTE_OFFLINE
The data of the file is not immediately available. Indicates that the file data has been physically moved to offline storage.

FILE_ATTRIBUTE_READONLY
The file is read only. Applications can read the file but cannot write to it or delete it.

FILE_ATTRIBUTE_SYSTEM
The file is part of or is used exclusively by the operating system.

FILE_ATTRIBUTE_TEMPORARY
The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

Any combination of the following flags is acceptable for the dwFlagsAndAttributes parameter.

FILE_FLAG_WRITE_THROUGH
Instructs the system to write through any intermediate cache and go directly to disk. Windows can still cache write operations, but cannot lazily flush them.

FILE_FLAG_OVERLAPPED
Instructs the system to initialize the object, so that operations that take a significant amount of time to process return ERROR_IO_PENDING. When the operation is finished, the specified event is set to the signaled state.

When you specify FILE_FLAG_OVERLAPPED, the ReadFile and WriteFile functions must specify an OVERLAPPED structure. That is, when FILE_FLAG_OVERLAPPED is specified, an application must perform overlapped reading and writing.

When FILE_FLAG_OVERLAPPED is specified, the system does not maintain the file pointer. The file position must be passed as part of the lpOverlapped parameter (pointing to an OVERLAPPED structure) to the ReadFile and WriteFile functions.

This flag also enables more than one operation to be performed simultaneously with the handle (a simultaneous read and write operation, for example).

FILE_FLAG_NO_BUFFERING
Instructs the system to open the file with no intermediate buffering or caching. When combined with FILE_FLAG_OVERLAPPED, the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations will take longer, because data is not being held in the cache.

An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

· File access must begin at byte offsets within the file that are integer multiples of the volume's sector size.

· File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.

· Buffer addresses for read and write operations must be aligned on addresses in memory that are integer multiples of the volume's sector size.

One way to align buffers on integer multiples of the volume sector size is to use VirtualAlloc to allocate the buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system's memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume's sector size.

An application can determine a volume's sector size by calling the GetDiskFreeSpace function.

FILE_FLAG_RANDOM_ACCESS
Indicates that the file is accessed randomly. The system can use this as a hint to optimize file caching.

FILE_FLAG_SEQUENTIAL_SCAN
Indicates that the file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed.

Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.

FILE_FLAG_DELETE_ON_CLOSE
Indicates that the operating system is to delete the file immediately after all of its handles have been closed, not just the handle for which you specified FILE_FLAG_DELETE_ON_CLOSE.

Subsequent open requests for the file will fail, unless FILE_SHARE_DELETE is used.

FILE_FLAG_BACKUP_SEMANTICS
Windows NT only: Indicates that the file is being opened or created for a backup or restore operation. The operating system ensures that the calling process overrides file security checks, provided it has the necessary permission to do so. The relevant permissions are SE_BACKUP_NAME and SE_RESTORE_NAME.

You can also set this flag to obtain a handle to a directory. A directory handle can be passed to some Win32 functions in place of a file handle.

FILE_FLAG_POSIX_SEMANTICS
Indicates that the file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support such naming. Use care when using this option because files created with this flag may not be accessible by applications written for MS-DOS or Windows.

If the CreateFile function opens the client side of a named pipe, the dwFlagsAndAttributes parameter can also contain Security Quality of Service information. When the calling application specifies the SECURITY_SQOS_PRESENT flag, the dwFlagsAndAttributes parameter can contain one or more of the following values:

SECURITY_ANONYMOUS
Specifies to impersonate the client at the Anonymous impersonation level.

SECURITY_IDENTIFICATION
Specifies to impersonate the client at the Identification impersonation level.

SECURITY_IMPERSONATION
Specifies to impersonate the client at the Impersonation impersonation level.

SECURITY_DELEGATION
Specifies to impersonate the client at the Delegation impersonation level.

SECURITY_CONTEXT_TRACKING
Specifies that the security tracking mode is dynamic. If this flag is not specified, Security Tracking Mode is static.

SECURITY_EFFECTIVE_ONLY
Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available.

This flag allows the client to limit the groups and privileges that a server can use while impersonating the client.

For more information, see Security.

hTemplateFile

Specifies a handle with GENERIC_READ access to a template file. The template file supplies file attributes and extended attributes for the file being created.

Windows 95: This value must be NULL. If you supply a handle under Windows 95, the call fails and GetLastError returns ERROR_NOT_SUPPORTED.

Return Values

If the function succeeds, the return value is an open handle to the specified file. If the specified file exists before the function call and dwCreationDistribution is CREATE_ALWAYS or OPEN_ALWAYS, a call to GetLastError returns ERROR_ALREADY_EXISTS (even though the function has succeeded). If the file does not exist before the call, GetLastError returns zero.

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

Remarks

Use the CloseHandle function to close an object handle returned by CreateFile.

As noted above, specifying zero for dwDesiredAccess allows an application to query device attributes without actually accessing the device. This type of querying is useful, for example, if an application wants to determine the size of a floppy disk drive and the formats it supports without having a floppy in the drive.

Files

When creating a new file, the CreateFile function performs the following actions:

· Combines the file attributes and flags specified by dwFlagsAndAttributes with FILE_ATTRIBUTE_ARCHIVE.

· Sets the file length to zero.

· Copies the extended attributes supplied by the template file to the new file if the hTemplateFile parameter is specified.

When opening an existing file, CreateFile performs the following actions:

· Combines the file flags specified by dwFlagsAndAttributes with existing file attributes. CreateFile ignores the file attributes specified by dwFlagsAndAttributes.

· Sets the file length according to the value of dwCreationDistribution.

· Ignores the hTemplateFile parameter.

· Ignores the lpSecurityDescriptor member of the SECURITY_ATTRIBUTES structure if the lpSecurityAttributes parameter is not NULL. The other structure members are used. The bInheritHandle member is the only way to indicate whether the file handle can be inherited.

If you are attempting to create a file on a floppy drive that does not have a floppy disk or a CD-ROM drive that does not have a CD, the system displays a message box asking the user to insert a disk or a CD, respectively. To prevent the system from displaying this message box, call the SetErrorMode function with SEM_FAILCRITICALERRORS.

Pipes

If CreateFile opens the client end of a named pipe, the function uses any instance of the named pipe that is in the listening state. The opening process can duplicate the handle as many times as required but, once opened, the named pipe instance cannot be opened by another client. The access specified when a pipe is opened must be compatible with the access specified in the dwOpenMode parameter of the CreateNamedPipe function. For more information about pipes, see Pipes.

Mailslots

If CreateFile opens the client end of a mailslot, the function returns INVALID_HANDLE_VALUE if the mailslot client attempts to open a local mailslot before the mailslot server has created it with the CreateMailSlot function. For more information about mailslots, see Mailslots.

Communications Resources

The CreateFile function can create a handle to a communications resource, such as the serial port COM1. For communications resources, the dwCreationDistribution parameter must be OPEN_EXISTING, and the hTemplate parameter must be NULL. Read, write, or read-write access can be specified, and the handle can be opened for overlapped I/O. For more information about communications, see Communications.

Disk Devices

Windows NT: You can use the CreateFile function to open a disk drive or a partition on a disk drive. The function returns a handle to the disk device; that handle can be used with the DeviceIOControl function. The following requirements must be met in order for such a call to succeed:

· The caller must have administrative privileges for the operation to succeed on a hard disk drive.

· The lpFileName string should be of the form \\.\PHYSICALDRIVEx to open the hard disk x. Hard disk numbers start at zero. For example:

\\.\PHYSICALDRIVE2
Obtains a handle to the third physical drive on the user's computer.

· The lpFileName string should be \\.\x: to open a floppy drive x or a partition x on a hard disk. For example:

\\.\A:
Obtains a handle to drive A on the user's computer.

\\.\C:
Obtains a handle to drive C on the user's computer.

Windows 95: This technique does not work for opening a logical drive. In Windows 95, specifying a string in this form causes CreateFile to return an error.

· The dwCreationDistribution parameter must have the OPEN_EXISTING value.

· When opening a floppy disk or a partition on a hard disk, you must set the FILE_SHARE_WRITE flag in the dwShareMode parameter.

Consoles

The CreateFile function can create a handle to console input (CONIN$). If the process has an open handle to it as a result of inheritance or duplication, it can also create a handle to the active screen buffer (CONOUT$). The calling process must be attached to an inherited console or one allocated by the AllocConsole function. For console handles, set the CreateFile parameters as follows:

lpFileName
Use the CONIN$ value to specify console input and the CONOUT$ value to specify console output.
CONIN$ gets a handle to the console's input buffer, even if the SetStdHandle function redirected the standard input handle. To get the standard input handle, use the GetStdHandle function.
CONOUT$ gets a handle to the active screen buffer, even if SetStdHandle redirected the standard output handle. To get the standard output handle, use GetStdHandle.

dwDesiredAccess
GENERIC_READ | GENERIC_WRITE is preferred, but either one can limit access.

dwShareMode
If the calling process inherited the console or if a child process should be able to access the console, this parameter must be FILE_SHARE_READ | FILE_SHARE_WRITE.

lpSecurityAttributes
If you want the console to be inherited, the bInheritHandle member of the SECURITY_ATTRIBUTES structure must be TRUE.

dwCreationDistribution
You should specify OPEN_EXISTING when using CreateFile to open the console.

dwFlagsAndAttributes
Ignored.

hTemplateFile

Ignored.

The following list shows the effects of various settings of fwdAccess and lpFileName.

lpFileName	fwdAccess	Result
CON	GENERIC_READ	Opens console for input.
CON	GENERIC_WRITE	Opens console for output.
CON	GENERIC_READ\GENERIC_WRITE	Windows 95: Causes CreateFile to fail; GetLastError returns ERROR_PATH_NOT_FOUND. Windows NT: Causes CreateFile to fail; GetLastError returns ERROR_FILE_NOT_FOUND.

Directories

An application cannot create a directory with CreateFile; it must call CreateDirectory or CreateDirectoryEx to create a directory.

Windows NT:

You can obtain a handle to a directory by setting the FILE_FLAG_BACKUP_SEMANTICS flag. A directory handle can be passed to some Win32 functions in place of a file handle.

Some file systems, such as NTFS, support compression for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression attribute of its parent directory.

See Also

AllocConsole, CloseHandle, ConnectNamedPipe, CreateDirectory, CreateDirectoryEx, CreateNamedPipe, DeviceIOControl, GetDiskFreeSpace, GetOverlappedResult, GetStdHandle, OpenFile, OVERLAPPED, ReadFile, SECURITY_ATTRIBUTES, SetErrorMode, SetStdHandle TransactNamedPipe

CreateWindow

The CreateWindow function creates an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu.

```
HWND CreateWindow(
LPCTSTR lpClassName,        // pointer to registered class name
LPCTSTR lpWindowName,       // pointer to window name
DWORD dwStyle,              // window style
int x,                      // horizontal position of window
int y,                      // vertical position of window
int nWidth,                 // window width
int nHeight,                // window height
HWND hWndParent,            // handle to parent or owner window
HMENU hMenu,                        // handle to menu or child-window identifier
HANDLE hInstance,           // handle to application instance
LPVOID lpParam                      // pointer to window-creation data
);
```

Parameters

lpClassName

Points to a null-terminated string or is an integer atom. If this parameter is an atom, it must be a global atom created by a previous call to the GlobalAddAtom function. The atom, a 16-bit value less than 0xC000, must be in the low-order word of lpClassName; the high-order word must be zero.

If lpClassName is a string, it specifies the window class name. The class name can be any name registered with the RegisterClass function or any of the predefined control-class names. For a complete list, see the following Remarks section.

lpWindowName

Points to a null-terminated string that specifies the window name.

dwStyle

Specifies the style of the window being created. This parameter can be a combination of the window styles and control styles listed in the following Remarks section.

x

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

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

y

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

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

nWidth

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

nHeight

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

hWndParent

Identifies the parent or owner window of the window being created. A valid window handle must be supplied when a child window or an owned window is created. A child window is confined to the client area of its parent window. An owned window is an overlapped window that is destroyed when its owner window is destroyed or hidden when its owner is minimized; it is always displayed on top of its owner window. Although this parameter must specify a valid handle if the dwStyle parameter includes the WS_CHILD style, it is optional if dwStyle includes the WS_POPUP style.

hMenu

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

hInstance

Identifies the instance of the module to be associated with the window.

lpParam

Points to a value passed to the window through the CREATESTRUCT structure referenced by the lParam parameter of the WM_CREATE message. If an application calls CreateWindow to create a multiple document interface (MDI) client window, lpParam must point to a CLIENTCREATESTRUCT structure.

Return Values

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

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

Remarks

Before returning, CreateWindow sends a WM_CREATE message to the window procedure.

For overlapped, pop-up, and child windows, CreateWindow sends WM_CREATE, WM_GETMINMAXINFO, and WM_NCCREATE messages to the window. The lParam parameter of the WM_CREATE message contains a pointer to a CREATESTRUCT structure. If the WS_VISIBLE style is specified, CreateWindow sends the window all the messages required to activate and show the window.

If the window style specifies a title bar, the window title pointed to by lpWindowName is displayed in the title bar. When using CreateWindow to create controls, such as buttons, check boxes, and static controls, use lpWindowName to specify the text of the control.

If you specify Windows version 4.x when linking your application, its windows cannot have caption buttons unless they also have window menus. This is not a requirement for applications that you linked specifying Windows version 3.x.

The following predefined control classes can be specified in the lpClassName parameter:

BUTTON
Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.

COMBOBOX
Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box.
Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field.

EDIT
Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the BACKSPACE key to delete characters.
Edit controls use the variable-pitch system font and display characters from the ANSI character set. The WM_SETFONT message can also be sent to the edit control to change the default font.
Edit controls expand tab characters into as many space characters as are required to move the caret to the next tab stop. Tab stops are assumed to be at every eighth character position.

LISTBOX
Designates a list of character strings. Specify this control whenever an application must present a list of names, such as filenames, from which the user can choose. The user can select a string by clicking it. A selected string is highlighted, and a notification message is passed to the parent window. Use a vertical or horizontal scroll bar with a list box to scroll lists that are too long for the control window. The list box automatically hides or shows the scroll bar, as needed.

MDICLIENT
Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD. Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view.

SCROLLBAR
Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a

notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. Scroll bar controls have the same appearance and function as scroll bars used in ordinary windows. Unlike scroll bars, however, scroll bar controls can be positioned anywhere in a window for use whenever scrolling input is needed for a window.
The scroll bar class also includes size box controls. A size box is a small rectangle the user can expand to change the size of the window.

STATIC
Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output.

The following window styles can be specified in the dwStyle parameter:

WS_BORDER
Creates a window that has a thin-line border.

WS_CAPTION
Creates a window that has a title bar (includes the WS_BORDER style).

WS_CHILD
Creates a child window. This style cannot be used with the WS_POPUP style.

WS_CHILDWINDOW
Same as the WS_CHILD style.

WS_CLIPCHILDREN
Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used when creating the parent window.

WS_CLIPSIBLINGS
Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.

WS_DISABLED
Creates a window that is initially disabled. A disabled window cannot receive input from the user.

WS_DLGFRAME
Creates a window that has a border of a style typically used with dialog boxes. A window with this style cannot have a title bar.

WS_GROUP
Specifies the first control of a group of controls. The group consists of this first control and all controls defined after it, up to the next control with the WS_GROUP style. The first control in each group usually has the WS_TABSTOP style so that the user can move from group to group. The user can subsequently change the keyboard focus from one control in the group to the next control in the group by using the direction keys.

WS_HSCROLL
Creates a window that has a horizontal scroll bar.

WS_ICONIC
Creates a window that is initially minimized. Same as the WS_MINIMIZE style.

WS_MAXIMIZE
Creates a window that is initially maximized.

WS_MAXIMIZEBOX
Creates a window that has a Maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.

WS_MINIMIZE
Creates a window that is initially minimized. Same as the WS_ICONIC style.

WS_MINIMIZEBOX
Creates a window that has a Minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.

WS_OVERLAPPED
Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.

WS_OVERLAPPEDWINDOW
Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.

WS_POPUP
Creates a pop-up window. This style cannot be used with the WS_CHILD style.

WS_POPUPWINDOW
Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.

WS_SIZEBOX
Creates a window that has a sizing border. Same as the WS_THICKFRAME style.

WS_SYSMENU
Creates a window that has a window-menu on its title bar. The WS_CAPTION style must also be specified.

WS_TABSTOP
Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style.

WS_THICKFRAME
Creates a window that has a sizing border. Same as the WS_SIZEBOX style.

WS_TILED
Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.

WS_TILEDWINDOW
Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style.

WS_VISIBLE
Creates a window that is initially visible.

WS_VSCROLL
Creates a window that has a vertical scroll bar.

The following button styles (in the BUTTON class) can be specified in the dwStyle parameter:

BS_3STATE
Creates a button that is the same as a check box, except that the box can be grayed as well as checked or unchecked. Use the grayed state to show that the state of the check box is not determined.

BS_AUTO3STATE
Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and unchecked.

BS_AUTOCHECKBOX
Creates a button that is the same as a check box, except that the check state automatically toggles between checked and unchecked each time the user selects the check box.

BS_AUTORADIOBUTTON
Creates a button that is the same as a radio button, except that when the user selects it, Windows automatically sets the button's check state to checked and automatically sets the check state for all other buttons in the same group to unchecked.

BS_CHECKBOX
Creates a small, empty check box with text. By default, the text is displayed to the right of the check box. To display the text to the left of the check box, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style).

BS_DEFPUSHBUTTON
Creates a push button that behaves like a BS_PUSHBUTTON style button, but also has a heavy black border. If the button is in a dialog box, the user can select the button by pressing the ENTER key, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely (default) option.

BS_GROUPBOX
Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper left corner.

BS_LEFTTEXT
Places text on the left side of the radio button or check box when combined with a radio button or check box style. Same as the BS_RIGHTBUTTON style.

BS_OWNERDRAW
Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created and a WM_DRAWITEM message when a visual aspect of the button has changed. Do not combine the BS_OWNERDRAW style with any other button styles.

BS_PUSHBUTTON
Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button.

BS_RADIOBUTTON
Creates a small circle with text. By default, the text is displayed to the right of the circle. To display the text to the left of the circle, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style). Use radio buttons for groups of related, but mutually exclusive choices.

BS_USERBUTTON
Obsolete, but provided for compatibility with 16-bit versions of Windows. Win32-based applications should use BS_OWNERDRAW instead.

BS_BITMAP
Specifies that the button displays a bitmap.

BS_BOTTOM
Places text at the bottom of the button rectangle.

BS_CENTER
Centers text horizontally in the button rectangle.

BS_ICON
Specifies that the button displays an icon.

BS_LEFT
Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button.

BS_MULTILINE
Wraps the button text to multiple lines if the text string is too long to fit on a single line in the button rectangle.

BS_NOTIFY
Enables a button to send BN_DBLCLK, BN_KILLFOCUS, and BN_SETFOCUS notification messages to its parent window. Note that buttons send the BN_CLICKED notification message regardless of whether it has this style.

BS_PUSHLIKE
Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.

BS_RIGHT
Right-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is right justified on the right side of the check box or radio button.

BS_RIGHTBUTTON
Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the BS_LEFTTEXT style.

BS_TEXT
Specifies that the button displays text.

BS_TOP
Places text at the top of the button rectangle.

BS_VCENTER
Places text in the middle (vertically) of the button rectangle.

The following combo box styles (in the COMBOBOX class) can be specified in the dwStyle parameter:

CBS_AUTOHSCROLL
Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.

CBS_DISABLENOSCROLL
Shows a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.

CBS_DROPDOWN
Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the edit

control.

CBS_DROPDOWNLIST
Similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item that displays the current selection in the list box.

CBS_HASSTRINGS
Specifies that an owner-drawn combo box contains items consisting of strings. The combo box maintains the memory and address for the strings, so the application can use the CB_GETLBTEXT message to retrieve the text for a particular item.

CBS_LOWERCASE
Converts to lowercase any uppercase characters entered into the edit control of a combo box.

CBS_NOINTEGRALHEIGHT
Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, Windows sizes a combo box so that it does not display partial items.

CBS_OEMCONVERT
Converts text entered in the combo box edit control. The text is converted from the Windows character set to the OEM character set and then back to the Windows set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN style.

CBS_OWNERDRAWFIXED
Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are all the same height. The owner window receives a WM_MEASUREITEM message when the combo box is created and a WM_DRAWITEM message when a visual aspect of the combo box has changed.

CBS_OWNERDRAWVARIABLE
Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a WM_MEASUREITEM message for each item in the combo box when you create the combo box; the owner window receives a WM_DRAWITEM message when a visual aspect of the combo box has changed.

CBS_SIMPLE
Displays the list box at all times. The current selection in the list box is displayed in the edit control.

CBS_SORT
Automatically sorts strings entered into the list box.

CBS_UPPERCASE
Converts to uppercase any lowercase characters entered into the edit control of a combo box.

The following edit control styles (in the EDIT class) can be specified in the dwStyle parameter:

ES_AUTOHSCROLL
Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.

ES_AUTOVSCROLL
Automatically scrolls text up one page when the user presses the ENTER key on the last line.

ES_CENTER

Centers text in a multiline edit control.

ES_LEFT
Left-aligns text.

ES_LOWERCASE
Converts all characters to lowercase as they are typed into the edit control.

ES_MULTILINE
Designates a multiline edit control. The default is single-line edit control.
When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the ES_WANTRETURN style.
When the multiline edit control is not in a dialog box and the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify ES_AUTOVSCROLL, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed.

If you specify the ES_AUTOHSCROLL style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify ES_AUTOHSCROLL, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed.

Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

ES_NOHIDESEL
Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus.

ES_NUMBER
Allows only digits to be entered into the edit control.

ES_OEMCONVERT
Converts text entered in the edit control. The text is converted from the Windows character set to the OEM character set and then back to the Windows set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.

ES_PASSWORD
Displays an asterisk (*) for each character typed into the edit control. You can use the EM_SETPASSWORDCHAR message to change the character that is displayed.

ES_READONLY
Prevents the user from typing or editing text in the edit control.

ES_RIGHT
Right-aligns text in a multiline edit control.

ES_UPPERCASE
Converts all characters to uppercase as they are typed into the edit control.

ES_WANTRETURN
Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline

edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.

The following list box control styles (in the LISTBOX class) can be specified in the dwStyle parameter:

LBS_DISABLENOSCROLL
Shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If you do not specify this style, the scroll bar is hidden when the list box does not contain enough items.

LBS_EXTENDEDSEL
Allows multiple items to be selected by using the SHIFT key and the mouse or special key combinations.

LBS_HASSTRINGS
Specifies that a list box contains items consisting of strings. The list box maintains the memory and addresses for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item. By default, all list boxes except owner-drawn list boxes have this style. You can create an owner-drawn list box either with or without this style.

LBS_MULTICOLUMN
Specifies a multicolumn list box that is scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.

LBS_MULTIPLESEL
Turns string selection on or off each time the user clicks or double-clicks a string in the list box. The user can select any number of strings.

LBS_NODATA
Specifies a no-data list box. Specify this style when the count of items in the list box will exceed one thousand. A no-data list box must also have the LBS_OWNERDRAWFIXED style, but must not have the LBS_SORT or LBS_HASSTRINGS style.

A no-data list box resembles an owner-drawn list box except that it contains no string or bitmap data for an item. Commands to add, insert, or delete an item always ignore any given item data; requests to find a string within the list box always fail. Windows sends the WM_DRAWITEM message to the owner window when an item must be drawn. The itemID member of the DRAWITEMSTRUCT structure passed with the WM_DRAWITEM message specifies the line number of the item to be drawn. A no-data list box does not send a WM_DELETEITEM message.

LBS_NOINTEGRALHEIGHT
Specifies that the size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.

LBS_NOREDRAW
Specifies that the list box's appearance is not updated when changes are made. You can change this style at any time by sending a WM_SETREDRAW message.

LBS_NOSEL
Specifies that the list box contains items that can be viewed but not selected.

LBS_NOTIFY
Notifies the parent window with an input message whenever the user clicks or double-clicks a string in the list box.

LBS_OWNERDRAWFIXED
Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are the same height. The owner window receives a WM_MEASUREITEM message when the list box is created and a WM_DRAWITEM message when a visual aspect of the list box has changed.

LBS_OWNERDRAWVARIABLE
Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a WM_MEASUREITEM message for each item in the combo box when the combo box is created and a WM_DRAWITEM message when a visual aspect of the combo box has changed.

LBS_SORT
Sorts strings in the list box alphabetically.

LBS_STANDARD
Sorts strings in the list box alphabetically. The parent window receives an input message whenever the user clicks or double-clicks a string. The list box has borders on all sides.

LBS_USETABSTOPS
Enables a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units. A dialog box unit is a horizontal or vertical distance. One horizontal dialog box unit is equal to one-fourth of the current dialog box base-width unit. Windows calculates these units based on the height and width of the current system font. The GetDialogBaseUnits function returns the current dialog box base units in pixels.

LBS_WANTKEYBOARDINPUT
Specifies that the owner of the list box receives WM_VKEYTOITEM messages whenever the user presses a key and the list box has the input focus. This enables an application to perform special processing on the keyboard input.

The following scroll bar styles (in the SCROLLBAR class) can be specified in the dwStyle parameter:

SBS_BOTTOMALIGN
Aligns the bottom edge of the scroll bar with the bottom edge of the rectangle defined by the parameters x, y, nWidth, and nHeight. The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style.

SBS_HORZ
Designates a horizontal scroll bar. If neither the SBS_BOTTOMALIGN nor SBS_TOPALIGN style is specified, the scroll bar has the height, width, and position defined by x, y, nWidth, and nHeight.

SBS_LEFTALIGN
Aligns the left edge of the scroll bar with the left edge of the rectangle defined by the parameters x, y, nWidth, and nHeight. The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style.

SBS_RIGHTALIGN
Aligns the right edge of the scroll bar with the right edge of the rectangle defined by the parameters x, y, nWidth, and nHeight. The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style.

SBS_SIZEBOX
Designates a size box. If you specify neither the SBS_SIZEBOXBOTTOMRIGHTALIGN nor the SBS_SIZEBOXTOPLEFTALIGN style, the size box has the height, width, and position specified by the parameters x, y, nWidth, and nHeight.

SBS_SIZEBOXBOTTOMRIGHTALIGN
Aligns the lower-right corner of the size box with the lower-right corner of the rectangle specified by the parameters x, y, nWidth, and nHeight. The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style.

SBS_SIZEBOXTOPLEFTALIGN
Aligns the upper-left corner of the size box with the upper-left corner of the rectangle specified by the parameters x, y, nWidth, and nHeight. The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style.

SBS_SIZEGRIP
Same as SBS_SIZEBOX, but with a raised edge.

SBS_TOPALIGN
Aligns the top edge of the scroll bar with the top edge of the rectangle defined by the parameters x, y, nWidth, and nHeight. The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style.

SBS_VERT
Designates a vertical scroll bar. If you specify neither the SBS_RIGHTALIGN nor the SBS_LEFTALIGN style, the scroll bar has the height, width, and position specified by the parameters x, y, nWidth, and nHeight.

The following static control styles (in the STATIC class) can be specified in the dwStyle parameter. A static control can have only one of these styles:

SS_BITMAP
Specifies a bitmap is to be displayed in the static control. The error code text is the name of a bitmap (not a filename) defined elsewhere in the resource file. The style ignores the nWidth and nHeight parameters; the control automatically sizes itself to accommodate the bitmap.

SS_BLACKFRAME
Specifies a box with a frame drawn in the same color as the window frames. This color is black in the default Windows color scheme.

SS_BLACKRECT
Specifies a rectangle filled with the current window frame color. This color is black in the default Windows color scheme.

SS_CENTER
Specifies a simple rectangle and centers the error code text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next centered line.

SS_CENTERIMAGE
Specifies that the midpoint of a static control with the SS_BITMAP or SS_ICON style is to remain fixed when the control is resized. The four sides are adjusted to accommodate a new bitmap or icon.
If a static control has the SS_BITMAP style and the bitmap is smaller than the control's client area, the client area is filled with the color of the pixel in the upper-left corner of the bitmap. If a static control has the SS_ICON style, the icon does not appear to paint the client area.

SS_GRAYFRAME
Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.

SS_GRAYRECT
Specifies a rectangle filled with the current screen background color. This color is gray in the default Windows color scheme.

SS_ICON
Specifies an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The style ignores the nWidth and nHeight parameters; the icon automatically sizes itself.

SS_LEFT
Specifies a simple rectangle and left-aligns the given text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next left-

aligned line.

SS_LEFTNOWORDWRAP
Specifies a simple rectangle and left-aligns the given text in the rectangle. Tabs are expanded but words are not wrapped. Text that extends past the end of a line is clipped.

SS_METAPICT
Specifies a metafile picture is to be displayed in the static control. The given text is the name of a metafile picture (not a filename) defined elsewhere in the resource file. A metafile static control has a fixed size; the metafile picture is scaled to fit the static control's client area.

SS_NOPREFIX
Prevents interpretation of any ampersand (&) characters in the control's text as accelerator prefix characters. These are displayed with the ampersand removed and the next character in the string underlined. This static control style may be included with any of the defined static controls.
An application can combine SS_NOPREFIX with other styles by using the bitwise OR (|) operator. This can be useful when filenames or other strings that may contain an ampersand (&) must be displayed in a static control in a dialog box.

SS_NOTIFY
Sends the parent window STN_CLICKED and STN_DBLCLK notification messages when the user clicks or double clicks the control.

SS_RIGHT
Specifies a simple rectangle and right-aligns the given text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next right-aligned line.

SS_RIGHTIMAGE
Specifies that the bottom-right corner of a static control with the SS_BITMAP or SS_ICON style is to remain fixed when the control is resized. Only the top and left sides are adjusted to accommodate a new bitmap or icon.

SS_SIMPLE
Specifies a simple rectangle and displays a single line of left-aligned text in the rectangle. The text line cannot be shortened or altered in any way. The control's parent window or dialog box must not process the WM_CTLCOLORSTATIC message.

SS_WHITEFRAME
Specifies a box with a frame drawn with the same color as the window backgrounds. This color is white in the default Windows color scheme.

SS_WHITERECT
Specifies a rectangle filled with the current window background color. This color is white in the default Windows color scheme.

The following dialog box styles can be specified in the dwStyle parameter:

DS_3DLOOK
Gives the dialog box a nonbold font and draws three-dimensional borders around control windows in the dialog box. The DS_3DLOOK style is required only by Win32-based applications compiled for versions of Windows earlier than Windows 95 or Windows NT 4.0. The system automatically applies the three-dimensional look to dialog boxes created by applications compiled for current versions of Windows.

DS_ABSALIGN
Indicates that the coordinates of the dialog box are screen coordinates; otherwise, Windows assumes they are client coordinates.

DS_CENTER
Centers the dialog box in the working area; that is, the area not obscured by the tray.

DS_CENTERMOUSE
Centers the mouse cursor in the dialog box.

DS_CONTEXTHELP
Includes a question mark in the title bar of the dialog box. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a control in the dialog box, the control receives a WM_HELP message. The control should pass the message to the dialog procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the control.
Note that DS_CONTEXTHELP is just a placeholder. When the dialog box is created, the system checks for DS_CONTEXTHELP and, if it is there, adds WS_EX_CONTEXTHELP to the extended style of the dialog box. WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.

DS_CONTROL
Creates a dialog box that works well as a child window of another dialog box, much like a page in a property sheet. This style allows the user to tab among the control windows of a child dialog box, use its accelerator keys, and so on.

DS_FIXEDSYS
Use SYSTEM_FIXED_FONT instead of SYSTEM_FONT.

DS_LOCALEDIT
Applies to 16-bit applications only. This style directs edit controls in the dialog box to allocate memory from the application's data segment. Otherwise, edit controls allocate storage from a global memory object.

DS_MODALFRAME
Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and window menu by specifying the WS_CAPTION and WS_SYSMENU styles.

DS_NOFAILCREATE
Creates the dialog box even if errors occur ¾ for example, if a child window cannot be created or if the system cannot create a special data segment for an edit control.

DS_NOIDLEMSG
Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.

DS_RECURSE
Dialog box style for control-like dialog boxes.

DS_SETFONT
Indicates that the dialog box template (the DLGTEMPLATE structure) contains two additional members specifying a font name and point size. The corresponding font is used to display text within the dialog box client area and within the dialog box controls. Windows passes the handle of the font to the dialog box and to each control by sending them the WM_SETFONT message.

DS_SETFOREGROUND
Does not apply to 16-bit versions of Microsoft Windows. This style brings the dialog box to the foreground. Internally, Windows calls the SetForegroundWindow function for the dialog box.

DS_SYSMODAL
Creates a system-modal dialog box. This style causes the dialog box to have the WS_EX_TOPMOST style, but

otherwise has no effect on the dialog box or the behavior of other windows in the system when the dialog box is displayed.

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

See Also

CharToOem, CLIENTCREATESTRUCT, CreateDialog, CREATESTRUCT, CreateWindowEx, DialogBox, DLGTEMPLATE, DRAWITEMSTRUCT, GetDialogBaseUnits, GlobalAddAtom, LB_GETTEXT, LB_SETCOLUMNWIDTH, MessageBox, RegisterClass, SetForegroundWindow, WM_COMMAND, WM_CREATE, WM_DELETEITEM, WM_DRAWITEM, WM_ENTERIDLE, WM_GETMINMAXINFO, WM_MEASUREITEM, WM_NCCREATE, WM_PAINT, WM_SETFONT

CreateWindowEx

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

```
HWND CreateWindowEx(
DWORD dwExStyle,            // extended window style
LPCTSTR lpClassName,        // pointer to registered class name
LPCTSTR lpWindowName,       // pointer to window name
DWORD dwStyle,              // window style
int x,                      // horizontal position of window
int y,                      // vertical position of window
int nWidth,                 // window width
int nHeight,                // window height
HWND hWndParent,            // handle to parent or owner window
HMENU hMenu,                        // handle to menu, or child-window identifier
HINSTANCE hInstance,        // handle to application instance
LPVOID lpParam                      // pointer to window-creation data
);
```

Parameters

dwExStyle

Specifies the extended style of the window. This parameter can be one of the following values:

WS_EX_ACCEPTFILES
Specifies that a window created with this style accepts drag-drop files.

WS_EX_APPWINDOW
Forces a top-level window onto the taskbar when the window is minimized.

WS_EX_CLIENTEDGE
Specifies that a window has a border with a sunken edge.

WS_EX_CONTEXTHELP
Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message. The child window should pass the message to the parent window procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the child window.
WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.

WS_EX_CONTROLPARENT
Allows the user to navigate among the child windows of the window by using the TAB key.

WS_EX_DLGMODALFRAME
Creates a window that has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.

WS_EX_LEFT
Window has generic "left-aligned" properties. This is the default.

WS_EX_LEFTSCROLLBAR
If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the vertical scroll bar (if present) is to the left of the client area. For other languages, the style is ignored and not treated as an error.

WS_EX_LTRREADING
The window text is displayed using Left to Right reading-order properties. This is the default.

WS_EX_MDICHILD
Creates an MDI child window.

WS_EX_NOPARENTNOTIFY
Specifies that a child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.

WS_EX_OVERLAPPEDWINDOW
Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.

WS_EX_PALETTEWINDOW
Combines the WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW, and WS_EX_TOPMOST styles.

WS_EX_RIGHT
Window has generic "right-aligned" properties. This depends on the window class. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored and not treated as an error.

WS_EX_RIGHTSCROLLBAR
Vertical scroll bar (if present) is to the right of the client area. This is the default.

WS_EX_RTLREADING
If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the window text is displayed using Right to Left reading-order properties. For other languages, the style is ignored and not treated as an error.

WS_EX_STATICEDGE
Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.

WS_EX_TOOLWINDOW
Creates a tool window; that is, a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB. If a tool window has a system menu, its icon is not displayed on the title bar. However, you can display the system menu by right-clicking or by typing ALT+SPACE.

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

WS_EX_TRANSPARENT
Specifies that a window created with this style is to be transparent. That is, any windows that are beneath the window are not obscured by the window. A window created with this style receives WM_PAINT messages only after all sibling windows beneath it have been updated.

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

Using the WS_EX_RIGHT style for static or edit controls has the same effect as using the SS_RIGHT or

ES_RIGHT style, respectively. Using this style with button controls has the same effect as using BS_RIGHT and BS_RIGHTBUTTON styles.

lpClassName

Points to a null-terminated string or is an integer atom. If lpClassName is an atom, it must be a global atom created by a previous call to GlobalAddAtom. The atom, a 16-bit value less than 0xC000, must be in the low-order word of lpClassName; the high-order word must be zero.

If lpClassName is a string, it specifies the window class name. The class name can be any name registered with the RegisterClass function or any of the predefined control-class names.

lpWindowName

Points to a null-terminated string that specifies the window name.

dwStyle

Specifies the style of the window being created. For a list of the window styles that can be specified in dwStyle, see CreateWindow.

x

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

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

y

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

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

nWidth

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

nHeight

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

hWndParent

Identifies the parent or owner window of the window being created. A valid window handle must be supplied when a child window or an owned window is created. A child window is confined to the client area of its parent window. An owned window is an overlapped window that is destroyed when its owner window is destroyed or hidden when its owner is minimized; it is always displayed on top of its owner window. Although this parameter must specify a valid handle if the dwStyle parameter includes the WS_CHILD style, it is optional if dwStyle includes the WS_POPUP style.

hMenu

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

hInstance

Identifies the instance of the module to be associated with the window.

lpParam

Points to a value passed to the window through the CREATESTRUCT structure referenced by the lParam parameter of the WM_CREATE message. If an application calls CreateWindow to create a multiple document interface client window, lpParam must point to a CLIENTCREATESTRUCT structure.

Return Values

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

If the function fails, the return value is NULL.

Remarks

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

For information about the window control classes, window styles, and control styles used with this function, see the description of the CreateWindow function.

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

See Also

CLIENTCREATESTRUCT, CREATESTRUCT, CreateWindow, GlobalAddAtom, RegisterClass, SetWindowPos, WM_CREATE, WM_NCCALCSIZE, WM_NCCREATE

DeleteFile

The DeleteFile function deletes an existing file.

```
BOOL DeleteFile(
LPCTSTR lpFileName // pointer to name of file to delete
);
```

Parameters

lpFileName

Points to a null-terminated string that specifies the file to be deleted.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If an application attempts to delete a file that does not exist, the DeleteFile function fails.

Windows 95: The DeleteFile function deletes a file even if it is open for normal I/O or as a memory-mapped file. To prevent loss of data, close files before attempting to delete them.

Windows NT: The DeleteFile function fails if an application attempts to delete a file that is open for normal I/O or as a memory-mapped file.

To close an open file, use the CloseHandle function.

See Also

CloseHandle

IsCharAlpha

The IsCharAlpha function determines whether a character is an alphabetic character. This determination is based on the semantics of the language selected by the user during setup or by using Control Panel.

```
BOOL IsCharAlpha(
TCHAR ch          // character to test
);
```

Parameters

ch

Specifies the character to be tested.

Return Values

If the character is alphabetic, the return value is nonzero.

If the character is not alphabetic, the return value is zero. To get extended error information, call GetLastError.

See Also

IsCharAlphaNumeric

IsCharAlphaNumeric

The IsCharAlphaNumeric function determines whether a character is either an alphabetic or a numeric character. This determination is based on the semantics of the language selected by the user during setup or by using Control Panel.

```
BOOL IsCharAlphaNumeric(
TCHAR ch          // character to test
);
```

Parameters

ch

Specifies the character to be tested.

Return Values

If the character is alphanumeric, the return value is nonzero.

If the character is not alphanumeric, the return value is zero. To get extended error information, call GetLastError.

See Also

IsCharAlpha

IsCharLower

The IsCharLower function determines whether a character is lowercase. This determination is based on the semantics of the language selected by the user during setup or by using Control Panel.

```
BOOL IsCharLower(
TCHAR ch          // character to test
);
```

Parameters

ch

Specifies the character to be tested.

Return Values

If the character is lowercase, the return value is nonzero.

If the character is not lowercase, the return value is zero. To get extended error information, call GetLastError.

See Also

IsCharUpper

IsCharUpper

The IsCharUpper function determines whether a character is uppercase. This determination is based on the semantics of the language selected by the user during setup or by using Control Panel.

```
BOOL IsCharUpper(
TCHAR ch          // character to test
);
```

Parameters

ch

Specifies the character to be tested.

Return Values

If the character is uppercase, the return value is nonzero.

If the character is not uppercase, the return value is zero. To get extended error information, call GetLastError.

See Also

IsCharLower

KillTimer

The KillTimer function destroys the specified timer.

```
BOOL KillTimer(
HWND hWnd,     // handle of window that installed timer
UINT uIDEvent  // timer identifier
);
```

Parameters

hWnd

Identifies the window associated with the specified timer. This value must be the same as the hWnd value passed to the SetTimer function that created the timer.

uIDEvent

Specifies the timer to be destroyed. If the window handle passed to SetTimer is valid, this parameter must be the same as the uIDEvent value passed to SetTimer. If the application calls SetTimer with hWnd set to NULL, this parameter must be the timer identifier returned by SetTimer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The KillTimer function does not remove WM_TIMER messages already posted to the message queue.

See Also

SetTimer

lstrcat

The lstrcat function appends one string to another.

```
LPTSTR lstrcat(
LPTSTR lpString1,        // address of buffer for concatenated strings
LPCTSTR lpString2        // address of string to add to string1
);
```

Parameters

lpString1

Points to a null-terminated string. The buffer must be large enough to contain both strings.

lpString2

Points to the null-terminated string to be appended to the string specified in the lpString1 parameter.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

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

See Also

~~lstrcmp~~

lstrcmp

The lstrcmp function compares two character strings. The comparison is case sensitive.

```
int lstrcmp(
LPCTSTR lpString1,      // address of first string
LPCTSTR lpString2       // address of second string
);
```

Parameters

lpString1

Points to the first null-terminated string to be compared.

lpString2

Points to the second null-terminated string to be compared.

Return Values

If the function succeeds and the string pointed to by lpString1 is less than the string pointed to by lpString2, the return value is negative; if the string pointed to by lpString1 is greater than the string pointed to by lpString2, it is positive. If the strings are equal, the return value is zero.

Remarks

The lstrcmp function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

The function returns the difference of the values of the first unequal characters it encounters. For example, lstrcmp determines that "abcz" is greater than "abcdefg" and returns the difference of z and d.

The language (locale) selected by the user at setup time, or via the control panel, determines which string is greater (or whether the strings are the same). If no language (locale) is selected, Windows performs the comparison by using default values.

With a double-byte character set (DBCS) version of Windows, this function can compare two DBCS strings.

The Win32 lstrcmp function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. Note that in 16-bit versions of Windows, lstrcmp uses a string sort. For a detailed discussion of word sorts and string sorts, see the Remarks section of the reference page for the CompareString function .

See Also

CompareString, lstrcmpi

lstrcmpi

The lstrcmpi function compares two character strings. The comparison is not case sensitive.

```
int lstrcmpi(
LPCTSTR lpString1,       // address of first string
LPCTSTR lpString2        // address of second string
);
```

Parameters

lpString1

Points to the first null-terminated string to be compared.

lpString2

Points to the second null-terminated string to be compared.

Return Values

If the function succeeds and the string pointed to by lpString1 is less than the string pointed to by lpString2, the return value is negative; if the string pointed to by lpString1 is greater than the string pointed to by lpString2, it is positive. If the strings are equal, the return value is zero.

Remarks

The lstrcmpi function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

The function returns the difference of the values of the first unequal characters it encounters. For example, lstrcmpi determines that "abcz" is greater than "abcdefg" and returns the difference of z and d.

The language (locale) selected by the user at setup time, or by using the control panel, determines which string is greater (or whether the strings are the same). If no language (locale) is selected, Windows performs the comparison by using default values.

For some locales, the lstrcmpi function may be insufficient. If this occurs, use CompareString to ensure proper comparison. For example, in Japan call CompareString with the IGNORE_CASE, IGNORE_KANATYPE, and IGNORE_WIDTH values to achieve the most appropriate non-exact string comparison. The IGNORE_KANATYPE and IGNORE_WIDTH values are ignored in non-Asian locales, so you can set these values for all locales and be guaranteed to have a culturally correct "insensitive" sorting regardless of the locale. Note that specifying these values slows performance, so use them only when necessary.

With a double-byte character set (DBCS) version of Windows, this function can compare two DBCS strings.

The Win32 lstrcmpi function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. Note that in 16-bit versions of Windows, lstrcmpi uses a string sort. For a detailed discussion of word sorts and string sorts, see the Remarks section of the reference page for the CompareString function .

See Also

CompareString, lstrcmp

lstrcpy

The lstrcpy function copies a string to a buffer.

```
LPTSTR lstrcpy(
LPTSTR lpString1,         // address of buffer
LPCTSTR lpString2         // address of string to copy
);
```

Parameters

lpString1

Points to a buffer to receive the contents of the string pointed to by the lpString2 parameter. The buffer must be large enough to contain the string, including the terminating null character.

lpString2

Points to the null-terminated string to be copied.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

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

Remarks

With a double-byte character set (DBCS) version of Windows, this function can be used to copy a DBCS string.

See Also

lstrcat, lstrcpyn

lstrcpyn

The lstrcpyn function copies a specified number of characters from a source string into a buffer.

```
LPTSTR lstrcpyn(
LPTSTR lpString1,        // address of target buffer
LPCTSTR lpString2,       // address of source string
int iMaxLength           // number of bytes or characters to copy
);
```

Parameters

lpString1

Points to a buffer into which the function copies characters. The buffer must be large enough to contain the number of bytes (ANSI version) or characters (Unicode version) specified by iMaxLength, including room for a terminating null character.

lpString2

Points to a null-terminated string from which the function copies characters.

iMaxLength

Specifies the number bytes (ANSI version) or characters (Unicode version) to be copied from the string pointed to by lpString2 into the buffer pointed to by lpString1, including a terminating null character.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

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

Remarks

Note that the buffer pointed to by lpString1 must be large enough to include a terminating null character, and the string length value specified by iMaxLength includes room for a terminating null character. Thus, the following code

```
TCHAR chBuffer[512] ;

lstrcpyn(chBuffer, "abcdefghijklmnop", 4) ;
```

... copies the string "abc", followed by a terminating null character, to chBuffer.

See Also

lstrcat, lstrcpy

lstrlen

The lstrlen function returns the length in bytes (ANSI version) or characters (Unicode version) of the specified string (not including the terminating null character).

```
int lstrlen(
LPCTSTR lpString            // address of string to count
);
```

Parameters

lpString

Points to a null-terminated string.

Return Values

If the function succeeds, the return value specifies the length of the string in bytes (ANSI version) or characters (Unicode version).

See Also

lstrcat, GetWindowTextLength

OpenFile

The OpenFile function creates, opens, reopens, or deletes a file.

This function is provided for compatibility with 16-bit versions of Windows. In particular, the OpenFile function cannot open a named pipe. Win32-based applications should use the CreateFile function.

```
HFILE OpenFile(
LPCSTR lpFileName,            // pointer to filename
LPOFSTRUCT lpReOpenBuff,      // pointer to buffer for file information
UINT uStyle                   // action and attributes
);
```

Parameters

lpFileName

Points to a null-terminated string that names the file to be opened. The string must consist of characters from the Windows 3.x character set. The OpenFile function does not support Unicode filenames.

lpReOpenBuff

Points to the OFSTRUCT structure that receives information about the file when it is first opened. The structure can be used in subsequent calls to the OpenFile function to refer to the open file.

The OFSTRUCT structure contains a pathname string member whose length is limited to OFS_MAXPATHNAME characters. OFS_MAXPATHNAME is currently defined to be 128. Because of this, you cannot use the OpenFile function to open a file whose path length exceeds 128 characters. The CreateFile function does not have such a path length limitation.

uStyle

Specifies the action to take. The following values can be combined by using the bitwise OR operator:

OF_CANCEL
Ignored. In the Win32 application programming interface (API), the OF_PROMPT style produces a dialog box containing a Cancel button.

OF_CREATE
Creates a new file. If the file already exists, it is truncated to zero length.

OF_DELETE
Deletes the file.

OF_EXIST
Opens the file and then closes it. Used to test for a file's existence.

OF_PARSE
Fills the OFSTRUCT structure but carries out no other action.

OF_PROMPT
Displays a dialog box if the requested file does not exist. The dialog box informs the user that Windows cannot find the file, and it contains Retry and Cancel buttons. Choosing the Cancel button directs OpenFile to return a file-not-found error message.

OF_READ
Opens the file for reading only.

OF_READWRITE
Opens the file for reading and writing.

OF_REOPEN
Opens the file using information in the reopen buffer.

OF_SHARE_COMPAT
For MS-DOS-based file systems using the Win32 API, opens the file with compatibility mode, allowing any process on a specified computer to open the file any number of times. Other efforts to open with any other sharing mode fail.

Windows NT: This flag is mapped to the CreateFile function's FILE_SHARE_READ | FILE_SHARE_WRITE flags.

OF_SHARE_DENY_NONE
Opens the file without denying read or write access to other processes. On MS-DOS-based file systems using the Win32 API, if the file has been opened in compatibility mode by any other process, the function fails.

Windows NT: This flag is mapped to the CreateFile function's FILE_SHARE_READ | FILE_SHARE_WRITE flags.

OF_SHARE_DENY_READ
Opens the file and denies read access to other processes. On MS-DOS-based file systems using the Win32 API, if the file has been opened in compatibility mode or for read access by any other process, the function fails. Windows NT: This flag is mapped to the CreateFile function's FILE_SHARE_WRITE flag.

OF_SHARE_DENY_WRITE
Opens the file and denies write access to other processes. On MS-DOS-based file systems using the Win32 API, if the file has been opened in compatibility mode or for write access by any other process, the function fails.

Windows NT: This flag is mapped to the CreateFile function's FILE_SHARE_READ flag.

OF_SHARE_EXCLUSIVE
Opens the file with exclusive mode, denying both read and write access to other processes. If the file has been opened in any other mode for read or write access, even by the current process, the function fails.

OF_VERIFY
Verifies that the date and time of the file are the same as when it was previously opened. This is useful as an extra check for read-only files.

OF_WRITE
Opens the file for writing only.

Return Values

If the function succeeds, the return value specifies a file handle.

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

Remarks

If the lpFileName parameter specifies a filename and extension only, this function searches for a matching file in the following directories, in the order shown:

1. The directory from which the application loaded.

2. The current directory.

3. Windows 95: The Windows system directory. Use the GetSystemDirectory function to get the path of this directory.

Windows NT: The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory. The name of this directory is SYSTEM32.

4. Windows NT: The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.

5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.

6. The directories that are listed in the PATH environment variable.

The lpFileName parameter cannot contain wildcard characters.

The Win32 OpenFile function does not support the OF_SEARCH flag supported by the 16-bit Windows OpenFile function. The OF_SEARCH flag directs Windows to search for a matching file even when the filename includes a full path. To search for a file in a Win32-based application, use the SearchPath function.

To close the file after use, call the _lclose function.

See Also

CreateFile, GetSystemDirectory, GetWindowsDirectory, _lclose

ReadFile

The ReadFile function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the read operation.

```
BOOL ReadFile(
HANDLE hFile,                       // handle of file to read
LPVOID lpBuffer,                    // address of buffer that receives data
DWORD nNumberOfBytesToRead,         // number of bytes to read
LPDWORD lpNumberOfBytesRead,        // address of number of bytes read
LPOVERLAPPED lpOverlapped           // address of structure for data
);
```

Parameters

hFile

Identifies the file to be read. The file handle must have been created with GENERIC_READ access to the file.

Windows NT

For asynchronous read operations, hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the CreateFile function, or a socket handle returned by the socket or accept functions.

Windows 95

For asynchronous read operations, hFile can be a communications resource, mailslot, or named pipe handle opened with the FILE_FLAG_OVERLAPPED flag by CreateFile, or a socket handle returned by the socket or accept functions. Windows 95 does not support asynchronous read operations on disk files.

lpBuffer

Points to the buffer that receives the data read from the file.

nNumberOfBytesToRead

Specifies the number of bytes to be read from the file.

lpNumberOfBytesRead

Points to the number of bytes read. ReadFile sets this value to zero before doing any work or error checking. If this parameter is zero when ReadFile returns TRUE on a named pipe, the other end of the message-mode pipe called the WriteFile function with nNumberOfBytesToWrite set to zero.

If lpOverlapped is NULL, lpNumberOfBytesRead cannot be NULL.

If lpOverlapped is not NULL, lpNumberOfBytesRead can be NULL. If this is an overlapped read operation, you can get the number of bytes read by calling GetOverlappedResult. If hFile is associated with an I/O completion port, you can get the number of bytes read by calling GetQueuedCompletionStatus.

lpOverlapped

Points to an OVERLAPPED structure. This structure is required if hFile was created with FILE_FLAG_OVERLAPPED.

If hFile was opened with FILE_FLAG_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid OVERLAPPED structure. If hFile was created with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the read operation is complete.

If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the OVERLAPPED structure and ReadFile may return before the read operation has been completed. In this case, ReadFile returns FALSE and the GetLastError function returns ERROR_IO_PENDING. This allows the calling process to continue while the read operation finishes. The event specified in the OVERLAPPED structure is set to the signaled state upon completion of the read operation.

If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the read operation starts at the current file position and ReadFile does not return until the operation has been completed.

If hFile is not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the OVERLAPPED structure. ReadFile does not return until the read operation has been completed.

Return Values

If the function succeeds, the return value is nonzero.

If the return value is nonzero and the number of bytes read is zero, the file pointer was beyond the current end of the file at the time of the read operation. However, if the file was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the return value is FALSE and GetLastError returns ERROR_HANDLE_EOF when the file pointer goes beyond the current end of file.

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

Remarks

ReadFile returns when one of the following is true: a write operation completes on the write end of the pipe, the number of bytes requested has been read, or an error occurs.

If part of the file is locked by another process and the read operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.

Characters can be read from the console input buffer by using ReadFile with a handle to console input. The console mode determines the exact behavior of the ReadFile function.

If a named pipe is being read in message mode and the next message is longer than the nNumberOfBytesToRead parameter specifies, ReadFile returns FALSE and GetLastError returns ERROR_MORE_DATA. The remainder of the message may be read by a subsequent call to the ReadFile or PeekNamedPipe function.

When reading from a communications device, the behavior of ReadFile is governed by the current communication timeouts as set and retrieved using the SetCommTimeouts and GetCommTimeouts functions. Unpredictable results can occur if you fail to set the timeout values. For more information about communication timeouts, see COMMTIMEOUTS.

If ReadFile attempts to read from a mailslot whose buffer is too small, the function returns FALSE and GetLastError returns ERROR_INSUFFICIENT_BUFFER.

If the anonymous write pipe handle has been closed and ReadFile attempts to read using the corresponding anonymous read pipe handle, the function returns FALSE and GetLastError returns ERROR_BROKEN_PIPE.

The ReadFile function may fail and return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY whenever there are too many outstanding asynchronous I/O requests.

The ReadFile code to check for the end-of-file condition (eof) differs for synchronous and asynchronous read operations.

When a synchronous read operation reaches the end of a file, ReadFile returns TRUE and sets *lpNumberOfBytesRead to zero. The following sample code tests for end-of-file for a synchronous read operation:

```
// Attempt a synchronous read operation.
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead, NULL) ;
// Check for end of file.
if (bResult &&   nBytesRead == 0, )
{
    // we're at the end of the file
}
```

An asynchronous read operation can encounter the end of a file during the initiating call to ReadFile, or during subsequent asynchronous operation.

If EOF is detected at ReadFile time for an asynchronous read operation, ReadFile returns FALSE and GetLastError returns ERROR_HANDLE_EOF.

If EOF is detected during subsequent asynchronous operation, the call to GetOverlappedResult to obtain the results of that operation returns FALSE and GetLastError returns ERROR_HANDLE_EOF.

To cancel all pending asynchronous I/O operations, use the CancelIO function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to read from a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the SetErrorMode function with SEM_NOOPENFILEERRORBOX.

The following sample code illustrates testing for end-of-file for an asynchronous read operation:

```
// set up overlapped structure fields
// to simplify this sample, we'll eschew an event handle
gOverLapped.Offset       = 0;
gOverLapped.OffsetHigh = 0;
gOverLapped.hEvent       = NULL;

// attempt an asynchronous read operation
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead,
    &gOverlapped) ;

// if there was a problem, or the async. operation's still pending ...
if (!bResult)
```

```
{
    // deal with the error code
    switch (dwError = GetLastError())
    {
        case ERROR_HANDLE_EOF:
        {
            // we're reached the end of the file
            // during the call to ReadFile

            // code to handle that
        }

        case ERROR_IO_PENDING:
        {
            // asynchronous i/o is still in progress

            // do something else for a while
            GoDoSomethingElse() ;

            // check on the results of the asynchronous read
            bResult = GetOverlappedResult(hFile, &gOverlapped,
                &nBytesRead, FALSE) ;

            // if there was a problem ...
            if (!bResult)
            {
                // deal with the error code
                switch (dwError = GetLastError())
                {
                    case ERROR_HANDLE_EOF:
                    {
                        // we're reached the end of the file
                        //during asynchronous operation
                    }

                    // deal with other error cases
                }
            }
        } // end case

        // deal with other error cases

    } // end switch
} // end if
```

See Also

CancelIo, CreateFile, GetCommTimeouts, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED, PeekNamedPipe, ReadFileEx, SetCommTimeouts

ReadFileEx

The ReadFileEx function reads data from a file asynchronously. It is designed solely for asynchronous operation, unlike the ReadFile function, which is designed for both synchronous and asynchronous operation. ReadFileEx lets an application perform other processing during a file read operation.

The ReadFileEx function reports its completion status asynchronously, calling a specified completion routine when reading is completed and the calling thread is in an alertable wait state.

```
BOOL ReadFileEx(
HANDLE hFile,                              // handle of file to read
LPVOID lpBuffer,                           // address of buffer
DWORD nNumberOfBytesToRead,                // number of bytes to read
LPOVERLAPPED lpOverlapped,                 // address of offset
LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // address of completion routine
);
```

Parameters

hFile

An open handle that specifies the file entity to be read from. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and must have GENERIC_READ access to the file.

Windows NT: hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the CreateFile function, or a socket handle returned by the socket or accept functions.

Windows 95: hFile can be a communications resource, mailslot, or named pipe handle opened with the FILE_FLAG_OVERLAPPED flag by CreateFile, or a socket handle returned by the socket or accept functions. Windows 95 does not support asynchronous operations on disk files.

lpBuffer

Points to a buffer that receives the data read from the file.

This buffer must remain valid for the duration of the read operation. The application should not use this buffer until the read operation is completed.

nNumberOfBytesToRead

Specifies the number of bytes to be read from the file.

lpOverlapped

Points to an OVERLAPPED data structure that supplies data to be used during the asynchronous (overlapped) file read operation.

If the file specified by hFile supports the concept of byte offsets, the caller of ReadFileEx must specify a byte offset within the file at which reading should begin. The caller specifies the byte offset by setting the OVERLAPPED structure's Offset and OffsetHigh members.

If the file entity specified by hFile does not support the concept of byte offsets ¾ for example, if it is a named pipe ¾ the caller must set the Offset and OffsetHigh members to zero, or ReadFileEx fails.

The ReadFileEx function ignores the OVERLAPPED structure's hEvent member. An application is free to use that member for its own purposes in the context of a ReadFileEx call. ReadFileEx signals completion of its read operation by calling, or queueing a call to, the completion routine pointed to by lpCompletionRoutine, so it does not need an event handle.

The ReadFileEx function does use the OVERLAPPED structure's Internal and InternalHigh members. An application should not set these members.

The OVERLAPPED data structure pointed to by lpOverlapped must remain valid for the duration of the read operation. It should not be a variable that can go out of scope while the file read operation is in progress.

lpCompletionRoutine

Points to the completion routine to be called when the read operation is complete and the calling thread is in an alertable wait state. For more information about the completion routine, see FileIOCompletionRoutine.

Return Values

If the function succeeds, the return value is nonzero.

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

If the function succeeds, the calling thread has an asynchronous I/O (input/output) operation pending: the overlapped read operation from the file. When this I/O operation completes, and the calling thread is blocked in an alertable wait state, the system calls the function pointed to by lpCompletionRoutine, and the wait state completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds, and the file reading operation completes, but the calling thread is not in an alertable wait state, the system queues the completion routine call, holding the call until the calling thread enters an alertable wait state. For information about alertable waits and overlapped input/output operations, see Synchronization and Overlapped Input and Output.

If ReadFileEx attempts to read past the end of the file, the function returns zero, and GetLastError returns ERROR_HANDLE_EOF.

Remarks

If a portion of the file specified by hFile is locked by another process, and the read operation specified in a call to ReadFileEx overlaps the locked portion, the call to ReadFileEx fails.

If ReadFileEx attempts to read data from a mailslot whose buffer is too small, the function returns FALSE, and GetLastError returns ERROR_INSUFFICIENT_BUFFER.

Applications must not read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.

The ReadFileEx function may fail if there are too many outstanding asynchronous I/O requests. In the event of such a failure, GetLastError can return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY.

To cancel all pending asynchronous I/O operations, use the CancelIO function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to read from a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the

SetErrorMode function with SEM_NOOPENFILEERRORBOX.

If hFile is a handle to a named pipe or other file entity that doesn't support the byte-offset concept, the Offset and OffsetHigh members of the OVERLAPPED structure pointed to by lpOverlapped must be zero, or ReadFileEx fails.

An application uses the MsgWaitForMultipleObjectsEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx, and SleepEx functions to enter an alertable wait state. For more information about alertable waits and overlapped input/output, refer to those functions' reference and Synchronization.

Windows 95: On this platform, neither ReadFileEx nor WriteFileEx can be used by the comm ports to communicate. However, you can use ReadFile and WriteFile to perform asynchronous communication.

See Also

CancelIo, CreateFile, FileIOCompletionRoutine, MsgWaitForMultipleObjectsEx, OVERLAPPED, ReadFile, SetErrorMode, SleepEx, WaitForMultipleObjectsEx, WaitForSingleObjectEx

RegCloseKey

The RegCloseKey function releases the handle of the specified key.

```
LONG RegCloseKey(
HKEY hKey        // handle of key to close
);
```

Parameters

hKey

Identifies the open key to close.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The handle for a specified key should not be used after it has been closed, because it will no longer be valid. Key handles should not be left open any longer than necessary.

The RegCloseKey function does not necessarily write information to the registry before returning; it can take as much as several seconds for the cache to be flushed to the hard disk. If an application must explicitly write registry information to the hard disk, it can use the RegFlushKey function. RegFlushKey, however, uses many system resources and should be called only when necessary.

See Also

RegCreateKey, RegCreateKeyEx, RegDeleteKey, RegFlushKey, RegOpenKey, RegOpenKeyEx, RegSetValue, RegSetValueEx

RegCreateKey

The RegCreateKey function creates the specified key. If the key already exists in the registry, the function opens it. This function is provided for compatibility with Windows version 3.1. Win32-based applications should use the RegCreateKeyEx function.

```
LONG RegCreateKey(
HKEY hKey,              // handle of an open key
LPCTSTR lpSubKey,       // address of name of subkey to open
PHKEY phkResult         // address of buffer for opened handle
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

The key opened or created by this function is a subkey of the key identified by hKey.

lpSubKey

Points to a null-terminated string specifying the name of a key that this function opens or creates. This key must be a subkey of the key identified by the hKey parameter.

If hKey is one of the predefined keys, lpSubKey may be NULL. In that case, the handle returned by using phkResult is the same hKey handle passed in to the function.

phkResult

Points to a variable that receives the handle of the opened or created key.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

An application can use the RegCreateKey function to create several keys at once. For example, an application can create a subkey four levels deep at the same time as the three preceding subkeys by specifying a string of the following form for the lpSubKey parameter:

subkey1\subkey2\subkey3\subkey4

The key identified by the hKey parameter must have been opened with KEY_CREATE_SUB_KEY access

(KEY_WRITE access includes KEY_CREATE_SUB_KEY access).

If the lpSubKey parameter is the address of an empty string, the function opens and then passes back the key identified by the hKey parameter.

See Also

RegCloseKey, RegCreateKeyEx, RegDeleteKey, RegOpenKey

RegCreateKeyEx

The RegCreateKeyEx function creates the specified key. If the key already exists in the registry, the function opens it.

```
LONG RegCreateKeyEx(
HKEY hKey,                    // handle of an open key
LPCTSTR lpSubKey,             // address of subkey name
DWORD Reserved,               // reserved
LPTSTR lpClass,                         // address of class string
DWORD dwOptions,              // special options flag
REGSAM samDesired,            // desired security access
LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // address of key security structure
PHKEY phkResult,              // address of buffer for opened handle
LPDWORD lpdwDisposition       // address of disposition value buffer
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

The key opened or created by the RegCreateKeyEx function is a subkey of the key identified by the hKey parameter.

lpSubKey

Points to a null-terminated string specifying the name of a subkey that this function opens or creates. The subkey specified must be a subkey of the key identified by the hKey parameter. This subkey must not begin with the backslash character ('\'). This parameter cannot be NULL.

Reserved

Reserved; must be zero.

lpClass

Points to a null-terminated string that specifies the class (object type) of this key. This parameter is ignored if the key already exists.

dwOptions

Specifies special options for the key. This parameter can be one of the following values.

REG_OPTION_NON_VOLATILE
This key is not volatile; this is the default. The information is stored in a file and is preserved when the system is restarted. The RegSaveKey function saves keys that are not volatile.

REG_OPTION_VOLATILE

Windows NT: This key is volatile; the information is stored in memory and is not preserved when the system is restarted. The RegSaveKey function does not save volatile keys. This flag is ignored if the key already exists.

Windows 95: This value is ignored in Windows 95. If REG_OPTION_VOLATILE is specified, the RegCreateKeyEx function creates a nonvolatile key and returns ERROR_SUCCESS.

REG_OPTION_BACKUP_RESTORE
Windows NT: If this flag is set, the function ignores the samDesired parameter and attempts to open the key with the access required to backup or restore the key. If the calling thread has the SE_BACKUP_NAME privilege enabled, the key is opened with ACCESS_SYSTEM_SECURITY and KEY_READ access. If the calling thread has the SE_RESTORE_NAME privilege enabled, the key is opened with ACCESS_SYSTEM_SECURITY and KEY_WRITE access. If both privileges are enabled, the key has the combined accesses for both privileges.

Windows 95: This flag is ignored. Windows 95 does not support security in its registry.

samDesired

Specifies an access mask that specifies the desired security access for the new key. This parameter can be a combination of the following values:

KEY_ALL_ACCESS
Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, KEY_CREATE_SUB_KEY, KEY_CREATE_LINK, and KEY_SET_VALUE access.

KEY_CREATE_LINK
Permission to create a symbolic link.

KEY_CREATE_SUB_KEY
Permission to create subkeys.

KEY_ENUMERATE_SUB_KEYS
Permission to enumerate subkeys.

KEY_EXECUTE
Permission for read access.

KEY_NOTIFY
Permission for change notification.

KEY_QUERY_VALUE
Permission to query subkey data.

KEY_READ
Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY access.

KEY_SET_VALUE
Permission to set subkey data.

KEY_WRITE
Combination of KEY_SET_VALUE and KEY_CREATE_SUB_KEY access.

lpSecurityAttributes

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

Windows NT: The lpSecurityDescriptor member of the structure specifies a security descriptor for the new key. If lpSecurityAttributes is NULL, the key gets a default security descriptor.

Windows 95: The lpSecurityDescriptor member of the structure is ignored.

phkResult

Points to a variable that receives the handle of the opened or created key.

lpdwDisposition

Points to a variable that receives one of the following disposition values:

REG_CREATED_NEW_KEY
The key did not exist and was created.

REG_OPENED_EXISTING_KEY
The key existed and was simply opened without being changed.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The key that the RegCreateKeyEx function creates has no values. An application can use the RegSetValue or RegSetValueEx function to set key values.

The key identified by the hKey parameter must have been opened with KEY_CREATE_SUB_KEY access. To open the key, use the RegCreateKeyEx or RegOpenKeyEx function.

An application cannot create a key under HKEY_USERS or HKEY_LOCAL_MACHINE.

An application can use RegCreateKeyEx to temporarily lock a portion of the registry. When the locking process creates a new key, it receives the disposition value REG_CREATED_NEW_KEY, indicating that it "owns" the lock. Another process attempting to create the same key receives the disposition value REG_OPENED_EXISTING_KEY, indicating that another process already owns the lock.

See Also

RegCloseKey, RegCreateKey, RegDeleteKey, RegOpenKey, RegOpenKeyEx

RegDeleteKey

Windows 95: The RegDeleteKey function deletes a key and all its descendents.

Windows NT: The RegDeleteKey function deletes the specified key. This function cannot delete a key that has subkeys.

```
LONG RegDeleteKey(
HKEY hKey,              // handle of open key
LPCTSTR lpSubKey        // address of name of subkey to delete
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

The key specified by the lpSubKey parameter must be a subkey of the key identified by hKey.

lpSubKey

Points to a null-terminated string specifying the name of the key to delete. This parameter cannot be NULL, and the specified key must not have subkeys.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

If the function succeeds, RegDeleteKey removes the specified key from the registry. The entire key, including all of its values, is removed.

To open the key, use the RegCreateKeyEx or RegOpenKeyEx function. Do not use the RegCreateKey or RegOpenKey functions.

See Also

RegCloseKey

RegDeleteValue

The RegDeleteValue function removes a named value from the specified registry key.

```
LONG RegDeleteValue(
HKEY hKey,                  // handle of key
LPCTSTR lpValueName         // address of value name
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpValueName

Points to a null-terminated string that names the value to remove. If this parameter is NULL or points to an empty string, the value set by the RegSetValue function is removed.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The key identified by the hKey parameter must have been opened with KEY_SET_VALUE access (KEY_WRITE access includes KEY_SET_VALUE access).

See Also

RegSetValue

RegLoadKey

The RegLoadKey function creates a subkey under HKEY_USER or HKEY_LOCAL_MACHINE and stores registration information from a specified file into that subkey. This registration information is in the form of a hive. A hive is a discrete body of keys, subkeys, and values that is rooted at the top of the registry hierarchy. A hive is backed by a single file and .LOG file.

```
LONG RegLoadKey(
HKEY hKey,                  // handle of open key
LPCTSTR lpSubKey,           // address of name of subkey
LPCTSTR lpFile              // address of filename for registry information
);
```

Parameters

hKey

Specifies the key where the subkey will be created. This can be a predefined reserved handle value, or a handle returned by a call to RegConnectRegistry. The predefined reserved handle values are:

HKEY_LOCAL_MACHINE
HKEY_USERS

This function always loads information at the top of the registry hierarchy. The HKEY_CLASSES_ROOT and HKEY_CURRENT_USER handle values cannot be specified for this parameter, because they represent subsets of the HKEY_LOCAL_MACHINE and HKEY_USERS handle values, respectively.

lpSubKey

Points to a null-terminated string that specifies the name of the key to be created under hKey. This subkey is where the registration information from the file will be loaded.

lpFile

Points to a null-terminated string containing the name of a file that has registration information. This file must have been created with the RegSaveKey function. Under the file allocation table (FAT) file system, the filename may not have an extension.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

If hKey is a handle returned by RegConnectRegistry, then the path specified in lpFile is relative to the remote computer.

Windows NT: The calling process must have the SE_RESTORE_NAME privilege. For more information about privileges, see Privileges.

Windows 95: Security privileges are not supported or required.

See Also

RegConnectRegistry, RegDeleteKey, RegReplaceKey, RegRestoreKey

RegOpenKey

The RegOpenKey function opens the specified key. This function is provided for compatibility with Windows version 3.1. Win32-based applications should use the RegOpenKeyEx function.

```
LONG RegOpenKey(
HKEY hKey,              // handle of open key
LPCTSTR lpSubKey,       // address of name of subkey to open
PHKEY phkResult         // address of handle of open key
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

The key opened by the RegOpenKey function is a subkey of the key identified by hKey.

lpSubKey

Points to a null-terminated string containing the name of the key to open. This key must be a subkey of the key identified by the hKey parameter. If this parameter is NULL or a pointer to an empty string, the function returns the same handle that was passed in.

phkResult

Points to a variable that receives the handle of the opened key.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The RegOpenKey function uses the default security access mask to open a key. If opening the key requires a different mask, the function fails, returning ERROR_ACCESS_DENIED. An application should use the RegOpenKeyEx function to specify an access mask in this situation.

Unlike the RegCreateKey function, RegOpenKey does not create the specified key if the key does not exist in the database.

See Also

RegCloseKey, RegCreateKey, RegCreateKeyEx, RegDeleteKey

RegOpenKeyEx

The RegOpenKeyEx function opens the specified key.

```
LONG RegOpenKeyEx(
HKEY hKey,                // handle of open key
LPCTSTR lpSubKey,         // address of name of subkey to open
DWORD ulOptions,          // reserved
REGSAM samDesired,        // security access mask
PHKEY phkResult           // address of handle of open key
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpSubKey

Points to a null-terminated string containing the name of the subkey to open. If this parameter is NULL or a pointer to an empty string, the function will open a new handle of the key identified by the hKey parameter. In this case, the function will not close the handles previously opened.

ulOptions

Reserved; must be zero.

samDesired

Specifies an access mask that describes the desired security access for the new key. This parameter can be a combination of the following values:

KEY_ALL_ACCESS
Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, KEY_CREATE_SUB_KEY, KEY_CREATE_LINK, and KEY_SET_VALUE access.

KEY_CREATE_LINK
Permission to create a symbolic link.

KEY_CREATE_SUB_KEY
Permission to create subkeys.

KEY_ENUMERATE_SUB_KEYS
Permission to enumerate subkeys.

KEY_EXECUTE
Permission for read access.

KEY_NOTIFY

Permission for change notification.

KEY_QUERY_VALUE
Permission to query subkey data.

KEY_READ
Combination of KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY access.

KEY_SET_VALUE
Permission to set subkey data.

KEY_WRITE
Combination of KEY_SET_VALUE and KEY_CREATE_SUB_KEY access.

phkResult

Points to a variable that receives the handle of the opened key.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

Unlike the RegCreateKeyEx function, the RegOpenKeyEx function does not create the specified key if the key does not exist in the registry.

See Also

RegCloseKey, RegCreateKeyEx, RegDeleteKey, RegOpenKey

RegQueryValue

The RegQueryValue function retrieves the value associated with the unnamed value for a specified key in the registry. Values in the registry have name, type, and data components. This function retrieves the data for a key's first value that has a NULL name. This function is provided for compatibility with Windows version 3.1. Win32-based applications should use the RegQueryValueEx function.

```
LONG RegQueryValue(
HKEY hKey,              // handle of key to query
LPCTSTR lpSubKey,       // address of name of subkey to query
LPTSTR lpValue,                  // address of buffer for returned string
PLONG lpcbValue         // address of buffer for size of returned string
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpSubKey

Points to a null-terminated string containing the name of the subkey of the hKey parameter for which a value is to be retrieved. If this parameter is NULL or points to an empty string, the function retrieves the value set by the RegSetValue function for the key identified by hKey.

lpValue

Points to a buffer that receives the value associated with the lpSubKey parameter. The buffer should be big enough to contain the terminating null character. This parameter can be NULL if the data is not required.

If lpValue is NULL, and lpcbValue is not NULL, the function places the size in bytes of the data referenced by the value key, including the terminating null character, into the variable pointed to by lpcbValue. This lets an application determine how to best preallocate a buffer for the value key's data.

lpcbValue

Points to a variable specifying the size, in bytes, of the buffer pointed to by the lpValue parameter. When the function returns, this variable contains the size of the data copied to lpValue, including the terminating null character.

If the buffer specified by lpValue parameter is not large enough to hold the data, the function returns the value ERROR_MORE_DATA, and stores the required buffer size, in bytes, into the variable pointed to by lpcbValue.

If lpValue is NULL, the function returns ERROR_SUCCESS, and stores the size of the string, in bytes, into the variable pointed to by lpcbValue. This lets an application determine the best way to allocate a buffer for the value key's data.

In all cases the value returned in lpcbValue always includes the size of the terminating null character in the string.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The key identified by the hKey parameter must have been opened with KEY_QUERY_VALUE access (KEY_READ access includes KEY_QUERY_VALUE access).

If the ANSI version of this function is used (either by explicitly calling RegQueryValue or by not defining Unicode before including the WINDOWS.H file), this function converts the stored Unicode string to an ANSI string before copying it to the buffer specified by the lpValue parameter.

See Also

RegEnumKey, RegEnumKeyEx, RegEnumValue, RegQueryInfoKey, RegQueryValueEx, RegSetValue, ~~RegSetValueEx~~

RegQueryValueEx

The RegQueryValueEx function retrieves the type and data for a specified value name associated with an open registry key.

```
LONG RegQueryValueEx(
HKEY hKey,                    // handle of key to query
LPTSTR lpValueName,           // address of name of value to query
LPDWORD lpReserved,           // reserved
LPDWORD lpType,               // address of buffer for value type
LPBYTE lpData,                // address of data buffer
LPDWORD lpcbData              // address of data buffer size
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpValueName

Points to a null-terminated string containing the name of the value to be queried.

lpReserved

Reserved; must be NULL.

lpType

Points to a variable that receives the key's value type. The value returned through this parameter will be one of the following:

REG_BINARY
Binary data in any form.

REG_DWORD
A 32-bit number.

REG_DWORD_LITTLE_ENDIAN
A 32-bit number in little-endian format (same as REG_DWORD). In little-endian format, the most significant byte of a word is the high-order byte. This is the most common format for computers running Windows NT and Windows 95.

REG_DWORD_BIG_ENDIAN
A 32-bit number in big-endian format. In big-endian format, the most significant byte of a word is the low-order byte.

REG_EXPAND_SZ

A null-terminated string that contains unexpanded references to environment variables (for example, "%PATH%"). It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.

REG_LINK
A Unicode symbolic link.

REG_MULTI_SZ
An array of null-terminated strings, terminated by two null characters.

REG_NONE
No defined value type.

REG_RESOURCE_LIST
A device-driver resource list.

REG_SZ
A null-terminated string. It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.

The lpType parameter can be NULL if the type is not required.

lpData

Points to a buffer that receives the value's data. This parameter can be NULL if the data is not required.

lpcbData

Points to a variable that specifies the size, in bytes, of the buffer pointed to by the lpData parameter. When the function returns, this variable contains the size of the data copied to lpData.

If the buffer specified by lpData parameter is not large enough to hold the data, the function returns the value ERROR_MORE_DATA, and stores the required buffer size, in bytes, into the variable pointed to by lpcbData.

If lpData is NULL, and lpcbData is non-NULL, the function returns ERROR_SUCCESS, and stores the size of the data, in bytes, in the variable pointed to by lpcbData. This lets an application determine the best way to allocate a buffer for the value key's data.

If the data has the REG_SZ, REG_MULTI_SZ or REG_EXPAND_SZ type, then lpData will also include the size of the terminating null character.

The lpcbData parameter can be NULL only if lpData is NULL.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

The key identified by hKey must have been opened with KEY_QUERY_VALUE access. To open the key, use the RegCreateKeyEx or RegOpenKeyEx function.

This function does not expand the environment-variable names in the value data when the value type is REG_EXPAND_SZ. The ExpandEnvironmentStrings function can be used to expand the environment-variable names.

If the value data has the REG_SZ, REG_MULTI_SZ or REG_EXPAND_SZ type, and the ANSI version of this function is used (either by explicitly calling RegQueryValueEx or by not defining Unicode before including the WINDOWS.H file), this function converts the stored Unicode string to an ANSI string before copying it to the buffer pointed to by lpData.

When calling the RegQueryValueEx function with hKey set to the HKEY_PERFORMANCE_DATA handle and a value string of a specified object, the returned data structure sometimes has unrequested objects. Don't be surprised; this is normal behavior. When calling the RegQueryValueEx function, you should always expect to walk the returned data structure to look for the requested object.

See Also

ExpandEnvironmentStrings, RegCreateKeyEx, RegEnumKey, RegEnumKeyEx, RegEnumValue, RegOpenKeyEx

RegSaveKey

The RegSaveKey function saves the specified key and all of its subkeys and values to a new file.

```
LONG RegSaveKey(
HKEY hKey,                    // handle of key where save begins
LPCTSTR lpFile,                            // address of filename to save to
LPSECURITY_ATTRIBUTES lpSecurityAttributes   // address of security structure
);
```

Parameters

hKey

Specifies a handle of the key where the save operation is to begin, or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpFile

Points to a null-terminated string containing the name of the file in which the specified key and subkeys are saved.

If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the RegLoadKey, RegReplaceKey, or RegRestoreKey function.

Windows NT: If the file already exists, the function fails with the ERROR_ALREADY_EXISTS error.

Windows 95: If the file already exists, the function fails with the ERROR_REGISTRY_IO_FAILED error.

Windows NT: If the string does not include a path, the file is created in the current directory of the calling process for a local key, or in the %systemroot%\system32 directory for a remote key.

Windows 95: If the string does not include a path, the file is created in the Windows root directory for local and remote keys. See GetWindowsDirectory.

lpSecurityAttributes

Windows NT: Pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new file. If lpSecurityAttributes is NULL, the file gets a default security descriptor.

Windows 95: This parameter is ignored.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

If hKey represents a key on a remote computer, the path described by lpFile is relative to the remote computer.

The RegSaveKey function saves only nonvolatile keys. It does not save volatile keys. A key is made volatile or nonvolatile at its creation; see RegCreateKeyEx.

Windows 95: The new file has the archive, hidden, readonly, and system attributes.

Windows NT: The new file has the archive attribute.

Windows NT: The calling process must have the SE_BACKUP_NAME privilege. For more information about privileges, see Privileges.

Windows 95: Security privileges are not supported or required.

See Also

RegCreateKeyEx, RegDeleteKey, RegLoadKey, RegReplaceKey

RegSetValue

The RegSetValue function associates a value with a specified key. This value must be a text string and cannot have a name. This function is provided for compatibility with Windows version 3.1. Win32-based applications should use the RegSetValueEx function, which allows an application to set any number of named values of any data type.

```
LONG RegSetValue(
HKEY hKey,              // handle of key to set value for
LPCTSTR lpSubKey,       // address of subkey name
DWORD dwType,           // type of value
LPCTSTR lpData,         // address of value data
DWORD cbData            // size of value data
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpSubKey

Points to a null-terminated string containing the name of the subkey with which a value is associated. This parameter can be null or a pointer to an empty string. In this case, the value will be added to the key identified by the hKey parameter.

dwType

Specifies the type of information to be stored. This parameter must be the REG_SZ type. To store other data types, use the RegSetValueEx function.

lpData

Points to a null-terminated string containing the value to set for the specified key.

cbData

Specifies the length, in bytes, of the string pointed to by the lpData parameter, not including the terminating null character.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

If the key specified by the lpSubKey parameter does not exist, the RegSetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the registry. This helps the registry perform efficiently.

The key identified by the hKey parameter must have been opened with KEY_SET_VALUE access. To open the key, use the RegCreateKeyEx or RegOpenKeyEx function. If the ANSI version of this function is used (either by explicitly calling RegSetValue or by not defining Unicode before including the WINDOWS.H file), the lpData parameter must be an ANSI character string. The string is converted to Unicode before it is stored in the registry.

See Also

RegCreateKeyEx, RegFlushKey, RegOpenKeyEx, ~~RegQueryValue~~

RegSetValueEx

The RegSetValueEx function stores data in the value field of an open registry key. It can also set additional value and type information for the specified key.

```
LONG RegSetValueEx(
HKEY hKey,                  // handle of key to set value for
LPCTSTR lpValueName,        // address of value to set
DWORD Reserved,             // reserved
DWORD dwType,               // flag for value type
CONST BYTE *lpData,         // address of value data
DWORD cbData                // size of value data
);
```

Parameters

hKey

Identifies a currently open key or any of the following predefined reserved handle values:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS

lpValueName

Points to a string containing the name of the value to set. If a value with this name is not already present in the key, the function adds it to the key.

If this parameter is NULL or points to an empty string and the dwType parameter is the REG_SZ type, this function sets the same value the RegSetValue function would set.

Reserved

Reserved; must be zero.

dwType

Specifies the type of information to be stored as the value's data. This parameter can be one of the following values:

REG_BINARY
Binary data in any form.

REG_DWORD
A 32-bit number.

REG_DWORD_LITTLE_ENDIAN
A 32-bit number in little-endian format (same as REG_DWORD). In little-endian format, the most significant byte of a word is the high-order byte. This is the most common format for computers running Windows NT and Windows 95.

REG_DWORD_BIG_ENDIAN
A 32-bit number in big-endian format. In big-endian format, the most significant byte of a word is the low-order

byte.

REG_EXPAND_SZ
A null-terminated string that contains unexpanded references to environment variables (for example, "%PATH%"). It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.

REG_LINK
A Unicode symbolic link.

REG_MULTI_SZ
An array of null-terminated strings, terminated by two null characters.

REG_NONE
No defined value type.

REG_RESOURCE_LIST
A device-driver resource list.

REG_SZ
A null-terminated string. It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.

lpData

Points to a buffer containing the data to be stored with the specified value name.

cbData

Specifies the size, in bytes, of the information pointed to by the lpData parameter. If the data is of type REG_SZ, REG_EXPAND_SZ, or REG_MULTI_SZ, cbData must include the size of the terminating null character.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT_MESSAGE_FROM_SYSTEM flag to get a generic description of the error.

Remarks

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the registry. This helps the registry perform efficiently. Application elements such as icons, bitmaps, and executable files should be stored as files and not be placed in the registry.

The key identified by the hKey parameter must have been opened with KEY_SET_VALUE access. To open the key, use the RegCreateKeyEx or RegOpenKeyEx function.

If dwType is the REG_SZ, REG_MULTI_SZ or REG_EXPAND_SZ type and the ANSI version of this function is used (either by explicitly calling RegSetValueEx or by not defining Unicode before including the WINDOWS.H file), the data pointed to by the lpData parameter must be an ANSI character string. The string is converted to Unicode before it is stored in the registry.

See Also

RegCreateKeyEx, RegFlushKey, RegOpenKeyEx, RegQueryValue

SetDlgItemInt

The SetDlgItemInt function sets the text of a control in a dialog box to the string representation of a specified integer value.

```
BOOL SetDlgItemInt(
HWND hDlg,       // handle of dialog box
int nIDDlgItem,  // identifier of control
UINT uValue,     // value to set
BOOL bSigned     // signed or unsigned indicator
);
```

Parameters

hDlg

Identifies the dialog box that contains the control.

nIDDlgItem

Specifies the control to be changed.

uValue

Specifies the integer value used to generate the item text.

bSigned

Specifies whether the uValue parameter is signed or unsigned. If this parameter is TRUE, uValue is signed. If this parameter is TRUE and uValue is less than zero, a minus sign is placed before the first digit in the string. If this parameter is FALSE, uValue is unsigned.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

To set the new text, this function sends a WM_SETTEXT message to the specified control.

See Also

GetDlgItemInt

SetDlgItemText

The SetDlgItemText function sets the title or text of a control in a dialog box.

```
BOOL SetDlgItemText(
HWND hDlg,              // handle of dialog box
int nIDDlgItem,         // identifier of control
LPCTSTR lpString        // text to set
);
```

Parameters

hDlg

Identifies the dialog box that contains the control.

nIDDlgItem

Identifies the control with a title or text that is to be set.

lpString

Points to the null-terminated string that contains the text to be copied to the control.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The SetDlgItemText function sends a WM_SETTEXT message to the specified control.

See Also

GetDlgItemInt, GetDlgItemText

SetFileAttributes

The SetFileAttributes function sets a file's attributes.

```
BOOL SetFileAttributes(
LPCTSTR lpFileName,            // address of filename
DWORD dwFileAttributes         // attributes to set
);
```

Parameters

lpFileName

Points to a string that specifies the name of the file whose attributes are to be set.

Windows 95: This string must not exceed MAX_PATH characters.

Windows NT: There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the SetFileAttributes function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of SetFileAttributes and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with SetFileAttributesW. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\wow\hotstuff\coolapps" is seen as "\\wow\hotstuff\coolapps".

dwFileAttributes

Specifies the file attributes to set for the file. This parameter can be a combination of the following values. However, all other values override FILE_ATTRIBUTE_NORMAL.

FILE_ATTRIBUTE_ARCHIVE
The file is an archive file. Applications use this value to mark files for backup or removal.

FILE_ATTRIBUTE_HIDDEN
The file is hidden. It is not included in an ordinary directory listing.

FILE_ATTRIBUTE_NORMAL
The file has no other attributes set. This value is valid only if used alone.

FILE_ATTRIBUTE_OFFLINE
The data of the file is not immediately available. Indicates that the file data has been physically moved to offline storage.

FILE_ATTRIBUTE_READONLY
The file is read-only. Applications can read the file but cannot write to it or delete it.

FILE_ATTRIBUTE_SYSTEM
The file is part of the operating system or is used exclusively by it.

FILE_ATTRIBUTE_TEMPORARY
The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

You cannot use the SetFileAttribute function to set a file's compression state. Setting FILE_ATTRIBUTE_COMPRESSED in the dwFileAttributes parameter does nothing. Use the DeviceIoControl function and the FSCTL_SET_COMPRESSION operation to set a file's compression state.

See Also

DeviceIoControl, FSCTL_SET_COMPRESSION, GetFileAttributes

SetWindowText

The SetWindowText function changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed.

```
BOOL SetWindowText(
HWND hWnd,              // handle of window or control
LPCTSTR lpString        // address of string
);
```

Parameters

hWnd

Identifies the window or control whose text is to be changed.

lpString

Points to a null-terminated string to be used as the new title or control text.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The SetWindowText function causes a WM_SETTEXT message to be sent to the specified window or control. If the window is a list box control created with the WS_CAPTION style, however, SetWindowText sets the text for the control, not for the list box entries.

The SetWindowText function does not expand tab characters (ASCII code 0x09). Tab characters are displayed as vertical bar (|) characters.

See Also

GetWindowText

ToAscii

The ToAscii function translates the specified virtual-key code and keyboard state to the corresponding Windows character or characters. The function translates the code using the input language and physical keyboard layout identified by the given keyboard layout handle.

```
int ToAscii(
UINT uVirtKey,          // virtual-key code
UINT uScanCode,         // scan code
PBYTE lpKeyState,       // address of key-state array
LPWORD lpChar,          // buffer for translated key
UINT uFlags             // active-menu flag
);
```

Parameters

uVirtKey

Specifies the virtual-key code to be translated.

uScanCode

Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

lpKeyState

Points to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the Caps Lock key is relevant. The toggle state of the Num Lock and Scroll Lock keys is ignored.

lpChar

Points to the buffer that will receive the translated Windows character or characters.

uFlags

Specifies whether a menu is active. This parameter must be 1 if a menu is active, or 0 otherwise.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values:

0
The specified virtual key has no translation for the current state of the keyboard.

1
One Windows character was copied to the buffer.

2
Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The parameters supplied to the ToAscii function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, ToAscii performs the translation based on the virtual-key code. In some cases, however, bit 15 of the uScanCode parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+number key combinations.

Although Num Lock is a toggle key that affects keyboard behavior, ToAscii ignores the toggle setting (the low bit) of lpKeyState (VK_NUMLOCK, because the uVirtKey parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0 - VK_NUMPAD9).

See Also

OemKeyScan, ToAsciiEx

ToAsciiEx

The ToAsciiEx function translates the specified virtual-key code and keyboard state to the corresponding Windows character or characters. The function translates the code using the input language and physical keyboard layout identified by the given keyboard layout handle.

```
int ToAsciiEx(
UINT uVirtKey,          // virtual-key code
UINT uScanCode,         // scan code
PBYTE lpKeyState,       // address of key-state array
LPWORD lpChar,          // buffer for translated key
UINT uFlags,            // active-menu flag
HKL dwhkl               // keyboard layout handle
);
```

Parameters

uVirtKey

Specifies the virtual-key code to be translated.

uScanCode

Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

lpKeyState

Points to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the Caps Lock key is relevant. The toggle state of the Num Lock and Scroll Lock keys is ignored.

lpChar

Points to the buffer that will receive the translated Windows character or characters.

uFlags

Specifies whether a menu is active. This parameter must be 1 if a menu is active, zero otherwise.

dwhkl

Identifies the keyboard layout to use to translate the given code. This parameter can be any keyboard layout handle previously returned by the LoadKeyboardLayout function.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values:

0
The specified virtual key has no translation for the current state of the keyboard.

1
One Windows character was copied to the buffer.

2
Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The parameters supplied to the ToAsciiEx function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, ToAsciiEx performs the translation based on the virtual-key code. In some cases, however, bit 15 of the uScanCode parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+number key combinations.

Although Num Lock is a toggle key that affects keyboard behavior, ToAsciiEx ignores the toggle setting (the low bit) of lpKeyState (VK_NUMLOCK, because the uVirtKey parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0 - VK_NUMPAD9).

See Also

LoadKeyboardLayout, MapVirtualKeyEx, OemKeyScan, ToAscii

WriteFile

The WriteFile function writes data to a file and is designed for both synchronous and asynchronous operation. The function starts writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with FILE_FLAG_OVERLAPPED. If the file handle was created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the write operation is finished.

```
BOOL WriteFile(
HANDLE hFile,                   // handle to file to write to
LPCVOID lpBuffer,               // pointer to data to write to file
DWORD nNumberOfBytesToWrite,            // number of bytes to write
LPDWORD lpNumberOfBytesWritten,         // pointer to number of bytes written
LPOVERLAPPED lpOverlapped    // pointer to structure needed for overlapped I/O
);
```

Parameters

hFile

Identifies the file to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

Windows NT

For asynchronous write operations, hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the CreateFile function, or a socket handle returned by the socket or accept functions.

Windows 95

For asynchronous write operations, hFile can be a communications resource, mailslot, or named pipe handle opened with the FILE_FLAG_OVERLAPPED flag by CreateFile, or a socket handle returned by the socket or accept functions. Windows 95 does not support asynchronous write operations on disk files.

lpBuffer

Points to the buffer containing the data to be written to the file.

nNumberOfBytesToWrite

Specifies the number of bytes to write to the file.

Unlike the MS-DOS operating system, Windows NT interprets a value of zero as specifying a null write operation. A null write operation does not write any bytes but does cause the time stamp to change.

Named pipe write operations across a network are limited to 65535 bytes.

lpNumberOfBytesWritten

Points to the number of bytes written by this function call. WriteFile sets this value to zero before doing any work or error checking.

If lpOverlapped is NULL, lpNumberOfBytesWritten cannot be NULL.

If lpOverlapped is not NULL, lpNumberOfBytesWritten can be NULL. If this is an overlapped write operation, you can get the number of bytes written by calling GetOverlappedResult. If hFile is associated with an I/O completion port, you can get the number of bytes written by calling GetQueuedCompletionStatus.

lpOverlapped

Points to an OVERLAPPED structure. This structure is required if hFile was opened with FILE_FLAG_OVERLAPPED.

If hFile was opened with FILE_FLAG_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid OVERLAPPED structure. If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the write operation is complete.

If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the write operation starts at the offset specified in the OVERLAPPED structure and WriteFile may return before the write operation has been completed. In this case, WriteFile returns FALSE and the GetLastError function returns ERROR_IO_PENDING. This allows the calling process to continue processing while the write operation is being completed. The event specified in the OVERLAPPED structure is set to the signaled state upon completion of the write operation.

If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the write operation starts at the current file position and WriteFile does not return until the operation has been completed.

If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the write operation starts at the offset specified in the OVERLAPPED structure and WriteFile does not return until the write operation has been completed.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If part of the file is locked by another process and the write operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.

Characters can be written to the screen buffer using WriteFile with a handle to console output. The exact behavior of the function is determined by the console mode. The data is written to the current cursor position. The cursor position is updated after the write operation.

Unlike the MS-DOS operating system, Windows NT interprets zero bytes to write as specifying a null write operation and WriteFile does not truncate or extend the file. To truncate or extend a file, use the SetEndOfFile function.

When writing to a nonblocking, byte-mode pipe handle with insufficient buffer space, WriteFile returns TRUE with *lpNumberOfBytesWritten < nNumberOfBytesToWrite.

When an application uses the WriteFile function to write to a pipe, the write operation may not finish if the pipe buffer is full. The write operation is completed when a read operation (using the ReadFile function) makes more buffer space available.

If the anonymous read pipe handle has been closed and WriteFile attempts to write using the corresponding anonymous write pipe handle, the function returns FALSE and GetLastError returns ERROR_BROKEN_PIPE.

The WriteFile function may fail with ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY whenever there are too many outstanding asynchronous I/O requests.

To cancel all pending asynchronous I/O operations, use the CancelIO function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to write to a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the SetErrorMode function with SEM_NOOPENFILEERRORBOX.

If hFile is a handle to a named pipe, the Offset and OffsetHigh members of the OVERLAPPED structure pointed to by lpOverlapped must be zero, or the function will fail.

See Also

CancelIo, CreateFile, GetLastError, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED, ReadFile, SetEndOfFile, WriteFileEx

WriteFileEx

The WriteFileEx function writes data to a file. It is designed solely for asynchronous operation, unlike WriteFile, which is designed for both synchronous and asynchronous operation.

WriteFileEx reports its completion status asynchronously, calling a specified completion routine when writing is completed and the calling thread is in an alertable wait state.

```
BOOL WriteFileEx(
HANDLE hFile,              // handle to output file
LPCVOID lpBuffer,          // pointer to input buffer
DWORD nNumberOfBytesToWrite,          // number of bytes to write
LPOVERLAPPED lpOverlapped,            // pointer to async. i/o data
LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine  // ptr. to completion routine
);
```

Parameters

hFile

An open handle that specifies the file entity to be written to. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and with GENERIC_WRITE access to the file.

Windows NT: hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the CreateFile function, or a socket handle returned by the socket or accept functions.

Windows 95: hFile can be a communications resource, mailslot, or named pipe handle opened with the FILE_FLAG_OVERLAPPED flag by CreateFile, or a socket handle returned by the socket or accept functions. Windows 95 does not support asynchronous operations on disk files.

lpBuffer

Points to the buffer containing the data to be written to the file.

This buffer must remain valid for the duration of the write operation. The caller must not use this buffer until the write operation is completed.

nNumberOfBytesToWrite

Specifies the number of bytes to write to the file.

If nNumberOfBtyesToWrite is zero, this function does nothing; in particular, it does not truncate the file. For additional discussion, see the following Remarks section.

lpOverlapped

Points to an OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

For files that support byte offsets, you must specify a byte offset at which to start writing to the file. You specify this offset by setting the Offset and OffsetHigh members of the OVERLAPPED structure. For files that do not support byte offsets ¾ named pipes, for example ¾ you must set Offset and OffsetHigh to zero, or WriteFileEx fails.

The WriteFileEx function ignores the OVERLAPPED structure's hEvent member. An application is free to use that member for its own purposes in the context of a WriteFileEx call. WriteFileEx signals completion of its writing operation by calling, or queueing a call to, the completion routine pointed to by lpCompletionRoutine, so it does not need an event handle.

The WriteFileEx function does use the Internal and InternalHigh members of the OVERLAPPED structure. You should not change the value of these members.

The OVERLAPPED data structure must remain valid for the duration of the write operation. It should not be a variable that can go out of scope while the write operation is pending completion.

lpCompletionRoutine

Points to a completion routine to be called when the write operation has been completed and the calling thread is in an alertable wait state. For more information about this completion routine, see FileIOCompletionRoutine.

Return Values

If the function succeeds, the return value is nonzero.

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

If the WriteFileEx function succeeds, the calling thread has an asynchronous I/O (input/output) operation pending: the overlapped write operation to the file. When this I/O operation finishes, and the calling thread is blocked in an alertable wait state, the operating system calls the function pointed to by lpCompletionRoutine, and the wait completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds and the file-writing operation finishes, but the calling thread is not in an alertable wait state, the system queues the call to *lpCompletionRoutine, holding the call until the calling thread enters an alertable wait state. See Synchronization for more information about alertable wait states and overlapped input/output operations.

Remarks

If part of the output file is locked by another process, and the specified write operation overlaps the locked portion, the WriteFileEx function fails.

Applications must not read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.

The WriteFileEx function may fail, returning the messages ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY if there are too many outstanding asynchronous I/O requests.

To cancel all pending asynchronous I/O operations, use the CancelIO function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to write to a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the SetErrorMode function with SEM_NOOPENFILEERRORBOX.

If hFile is a handle to a named pipe, or other file entity that doesn't support byte offsets, the Offset and OffsetHigh members of the OVERLAPPED structure pointed to by lpOverlapped must be zero, or the WriteFileEx function fails.

An application uses the WaitForSingleObjectEx, WaitForMultipleObjectsEx, MsgWaitForMultipleObjectsEx, SignalObjectAndWait, and SleepEx functions to enter an alertable wait state. Refer to Synchronization for more information about alertable wait states and overlapped input/output operations.

Windows 95: On this platform, neither WriteFileEx nor ReadFileEx can be used by the comm ports to communicate. However, you can use WriteFile and ReadFile to perform asynchronous communication.

See Also

CancelIo, CreateFile, FileIOCompletionRoutine, MsgWaitForMultipleObjectsEx, OVERLAPPED, ReadFileEx, SetEndOfFile, SetErrorMode, SleepEx, SignalObjectAndWait, WaitForMultipleObjectsEx

WritePrivateProfileString

The WritePrivateProfileString function copies a string into the specified section of the specified initialization file.

This function is provided for compatibility with 16-bit Windows-based applications. WIn32-based applications should store initialization information in the registry.

```
BOOL WritePrivateProfileString(
LPCTSTR lpAppName,              // pointer to section name
LPCTSTR lpKeyName,              // pointer to key name
LPCTSTR lpString,               // pointer to string to add
LPCTSTR lpFileName              // pointer to initialization filename
);
```

Parameters

lpAppName

Points to a null-terminated string containing the name of the section to which the string will be copied. If the section does not exist, it is created. The name of the section is case-independent; the string can be any combination of uppercase and lowercase letters.

lpKeyName

Points to the null-terminated string containing the name of the key to be associated with a string. If the key does not exist in the specified section, it is created. If this parameter is NULL, the entire section, including all entries within the section, is deleted.

lpString

Points to a null-terminated string to be written to the file. If this parameter is NULL, the key pointed to by the lpKeyName parameter is deleted.

Windows 95: This platform does not support the use of the TAB (\t) character as part of this parameter.

lpFileName

Points to a null-terminated string that names the initialization file.

Return Values

If the function successfully copies the string to the initialization file, the return value is nonzero.

If the function fails, or if it flushes the cached version of the most recently accessed initialization file, the return value is zero. To get extended error information, call GetLastError.

Remarks

Windows 95:

Windows 95 keeps a cached version of WIN.INI to improve performance. If all three parameters are NULL, the function flushes the cache. The function always returns FALSE after flushing the cache, regardless of whether the flush succeeds or fails.

A section in the initialization file must have the following form:

```
[section]
key=string
        .
        .
        .
```

If the lpFileName parameter does not contain a full path and filename for the file, WritePrivateProfileString searches the Windows directory for the file. If the file does not exist, this function creates the file in the Windows directory.

If lpFileName contains a full path and filename and the file does not exist, WriteProfileString creates the file. The specified directory must already exist.

Windows NT:

Windows NT maps most .INI file references to the registry, using the mapping defined under the following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping

Windows NT keeps a cache for the IniFileMapping registry key. Calling WritePrivateProfileStringW with the value of all arguments set to NULL will cause Windows NT to refresh its cache of the IniFileMappingKey for the specified .INI file.

The Win32 Profile functions (Get/WriteProfile*, Get/WritePrivateProfile*) use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say myfile.ini, under IniFileMapping:

HKEY_LOCAL_MACHINE\Software\Microsoft\
Windows NT\CurrentVersion\IniFileMapping\myfile.ini

2. Look for the section name specified by lpAppName. This will be a named value under myfile.ini, or a subkey of myfile.ini, or will not exist.

3. If the section name specified by lpAppName is a named value under myfile.ini, then that value specifies where in the registry you will find the keys for the section.

4. If the section name specified by lpAppName is a subkey of myfile.ini, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as "<No Name>") that specifies the default location in the registry where you will find the key.

5. If the section name specified by lpAppName does not exist as a named value or as a subkey under myfile.ini, then there will be an unnamed value (shown as "<No Name>") under myfile.ini that specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey for myfile.ini, or if there is no entry for the section name, then look for the actual myfile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the ini file mapping:

! - this character forces all writes to go both to the registry and to the .INI file on disk.

# - this character causes the registry value to be set to the value in the Windows 3.1 .INI file when a new user logs in for the first time after setup.

@ - this character prevents any reads from going to the .INI file on disk if the requested data is not found in the registry.

USR: - this prefix stands for HKEY_CURRENT_USER, and the text after the prefix is relative to that key.

SYS: - this prefix stands for HKEY_LOCAL_MACHINE\SOFTWARE, and the text after the prefix is relative to that key.

An application using the WritePrivateProfileStringW function to enter .INI file information into the registry should follow these guidelines:

· Ensure that no .INI file of the specified name exists on the system.

· Ensure that there is a key entry in the registry that specifies the .INI file. This entry should be under the path HKEY_LOCAL_MACHINE\SOFTWARE \Microsoft\Windows NT\CurrentVersion\IniFileMapping.

· Specify a value for that .INI file key entry that specifies a section. That is to say, an application must specify a section name, as it would appear within an .INI file or registry entry. Here is an example: [My Section].

· For system files, specify SYS for an added value.

· For application files, specify USR within the added value. Here is an example: "My Section: USR: App Name\Section". And, since USR indicates a mapping under HKEY_CURRENT_USER, the application should also create a key under HKEY_CURRENT_USER that specifies the application name listed in the added value. For the example just given, that would be "App Name".

· After following the preceding steps, an application setup program should call WritePrivateProfileStringW with the first three parameters set to NULL, and the fourth parameter set to the INI filename. For example:

```
WritePrivateProfileStringW( NULL, NULL, NULL, L"appname.ini" );
```

· Such a call causes the mapping of an .INI file to the registry to take effect before the next system reboot. The operating system re-reads the mapping information into shared memory. A user will not have to reboot their computer after installing an application in order to have future invocations of the application see the mapping of the .INI file to the registry.

The following sample code illustrates the preceding guidelines and is based on several assumptions:

· There is an application named "App Name."

· That application uses an .INI file named "appname.ini."

· There is a section in the .INI file that we want to look like this:

```
[Section1]
  FirstKey = It all worked out okay.
```

```
   SecondKey = By golly, it works.
   ThirdKey = Another test.
```

· The user will not have to reboot the system in order to have future invocations of the application see the mapping of the .INI file to the registry.

Here is the sample code :

```
// include files
#include <stdio.h>
#include <windows.h>

// a main function
main()

{
   // local variables
   CHAR inBuf[80];
   HKEY   hKey1, hKey2;
   DWORD   dwDisposition;
   LONG     lRetCode;

   // try to create the .INI file key
   lRetCode = RegCreateKeyEx ( HKEY_LOCAL_MACHINE,
                                            "SOFTWARE\\Microsoft\\Windows NT
                                              \\CurrentVersion\\IniFileMapping\\appname.ini",
                                            0, NULL, REG_OPTION_NON_VOLATILE, KEY_WRITE,
                                            NULL, &hKey1,
                                            &dwDisposition);

   // if we failed, note it, and leave
   if (lRetCode != ERROR_SUCCESS){
      printf ("Error in creating appname.ini key\n");
      return (0) ;
      }

   // try to set a section value
   lRetCode = RegSetValueEx ( hKey1,
                                          "Section1",
                                          0,
                                          REG_SZ,
                                          "USR:App Name\\Section1",
                                          20);

   // if we failed, note it, and leave
   if (lRetCode != ERROR_SUCCESS) {
      printf ( "Error in setting Section1 value\n");
      return (0) ;
      }

   // try to create an App Name key
   lRetCode = RegCreateKeyEx ( HKEY_CURRENT_USER,
                                            "App Name",
                                            0, NULL, REG_OPTION_NON_VOLATILE, KEY_WRITE,
                                                NULL, &hKey2,
                                                &dwDisposition);

   // if we failed, note it, and leave
   if (lRetCode != ERROR_SUCCESS) {
      printf ("Error in creating App Name key\n");
      return (0) ;
      }

   // force the operating system to re-read the mapping into shared memory
   //      so that future invocations of the application will see it
   //      without the user having to reboot the system
   WritePrivateProfileStringW( NULL, NULL, NULL, L"appname.ini" );

   // if we get this far, all has gone well
   // let's write some added values
   WritePrivateProfileString ("Section1", "FirstKey",
                                              "It all worked out okay.", "appname.ini");
   WritePrivateProfileString ("Section1", "SecondKey",
                                              "By golly, it works.", "appname.ini");
   WritePrivateProfileSection ("Section1", "ThirdKey = Another Test.",
                                                "appname.ini");

   // let's test our work
   GetPrivateProfileString ("Section1", "FirstKey",
                                           "Bogus Value: Get didn't work", inBuf, 80,
                                           "appname.ini");
   printf ("%s", inBuf);

   // okay, we are outta here
   return(0);

}
```

See Also

GetPrivateProfileString

MessageBeep

The MessageBeep function plays a waveform sound. The waveform sound for each sound type is identified by an entry in the [sounds] section of the registry.

```
BOOL MessageBeep(
UINT uType       // sound type
);
```

Parameters

uType

Specifies the sound type, as identified by an entry in the [sounds] section of the registry. This parameter can be one of the following values:

0xFFFFFFFF	Standard beep using the computer speaker
MB_ICONASTERISK	SystemAsterisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

After queuing the sound, the MessageBeep function returns control to the calling function and plays the sound asynchronously.

If it cannot play the specified alert sound, MessageBeep attempts to play the system default sound. If it cannot play the system default sound, the function produces a standard beep sound through the computer speaker.

The user can disable the warning beep by using the Control Panel Sound application.

See Also

FlashWindow, MessageBox, Beep

MessageBox

The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
int MessageBox(
HWND hWnd,            // handle of owner window
LPCTSTR lpText,       // address of text in message box
LPCTSTR lpCaption,    // address of title of message box
UINT uType            // style of message box
);
```

Parameters

hWnd

Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

Points to a null-terminated string containing the message to be displayed.

lpCaption

Points to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.

uType

Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Specify one of the following flags to indicate the buttons contained in the message box:

MB_ABORTRETRYIGNORE
The message box contains three push buttons: Abort, Retry, and Ignore.

MB_OK
The message box contains one push button: OK. This is the default.

MB_OKCANCEL
The message box contains two push buttons: OK and Cancel.

MB_RETRYCANCEL
The message box contains two push buttons: Retry and Cancel.

MB_YESNO
The message box contains two push buttons: Yes and No.

MB_YESNOCANCEL
The message box contains three push buttons: Yes, No, and Cancel.

Specify one of the following flags to display an icon in the message box:

MB_ICONEXCLAMATION,   MB_ICONWARNING
An exclamation-point icon appears in the message box.

MB_ICONINFORMATION, MB_ICONASTERISK
An icon consisting of a lowercase letter i in a circle appears in the message box.

MB_ICONQUESTION
A question-mark icon appears in the message box.

MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND
A stop-sign icon appears in the message box.

Specify one of the following flags to indicate the default button:

MB_DEFBUTTON1
The first button is the default button.

MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.

MB_DEFBUTTON2
The second button is the default button.

MB_DEFBUTTON3
The third button is the default button.

MB_DEFBUTTON4
The fourth button is the default button.

Specify one of the following flags to indicate the modality of the dialog box:

MB_APPLMODAL
The user must respond to the message box before continuing work in the window identified by the hWnd parameter. However, the user can move to the windows of other applications and work in those windows.

Depending on the hierarchy of windows in the application, the user may be able to move to other windows within the application. All child windows of the parent of the message box are automatically disabled, but popup windows are not.

MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.

MB_SYSTEMMODAL
Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with hWnd.

MB_TASKMODAL
Same as MB_APPLMODAL except that all the top-level windows belonging to the current task are disabled if the hWnd parameter is NULL. Use this flag when the calling application or library does not have a window handle available but still needs to prevent input to other windows in the current application without suspending other applications.

In addition, you can specify the following flags:

MB_DEFAULT_DESKTOP_ONLY
The desktop currently receiving input must be a default desktop; otherwise, the function fails. A default desktop is one an application runs on after the user has logged on.

MB_HELP
Adds a Help button to the message box. Choosing the Help button or pressing F1 generates a Help event.

MB_RIGHT
The text is right-justified.

MB_RTLREADING
Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems.

MB_SETFOREGROUND
The message box becomes the foreground window. Internally, Windows calls the SetForegroundWindow function for the message box.

MB_TOPMOST
The message box is created with the WS_EX_TOPMOST window style.

MB_SERVICE_NOTIFICATION
Windows NT only: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer.

If this flag is set, the hWnd parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the hWnd.

For Windows NT version 4.0, the value of MB_SERVICE_NOTIFICATION has changed. See WINUSER.H for the old and new values. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of MessageBox and MessageBoxEx. This mapping is only done for executables that have a version number, as set by the linker, less than 4.0.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you have two choices.

1. At link-time, specify a version number less than 4.0; or

2. At link-time, specify version 4.0. At run-time, use the GetVersionEx function to check the system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

MB_SERVICE_NOTIFICATION_NT3X
Windows NT only: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51.

Return Values

The return value is zero if there is not enough memory to create the message box.

If the function succeeds, the return value is one of the following menu-item values returned by the dialog box:

IDABORT
Abort button was selected.

IDCANCEL
Cancel button was selected.

IDIGNORE
Ignore button was selected.

IDNO
No button was selected.

IDOK
OK button was selected.

IDRETRY
Retry button was selected.

IDYES
Yes button was selected.

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the lpText and lpCaption parameters should not be taken from a resource file, because an attempt to load the resource may fail.

When an application calls MessageBox and specifies the MB_ICONHAND and MB_SYSTEMMODAL flags for the uType parameter, Windows displays the resulting message box regardless of available memory. When these flags are specified, Windows limits the length of the message box text to three lines. Windows does not automatically break the lines to fit in the message box, however, so the message string must contain carriage returns to break the lines at the appropriate places.

If you create a message box while a dialog box is present, use the handle of the dialog box as the hWnd parameter. The hWnd parameter should not identify a child window, such as a control in a dialog box.

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

See Also

FlashWindow, MessageBeep, MessageBoxEx, MessageBoxIndirect, SetForegroundWindow

SetTimer

The SetTimer function creates a timer with the specified time-out value.

```
UINT SetTimer(
HWND hWnd,    // handle of window for timer messages
UINT nIDEvent,              // timer identifier
UINT uElapse,   // time-out value
TIMERPROC lpTimerFunc // address of timer procedure
);
```

Parameters

hWnd

Identifies the window to be associated with the timer. This window must be owned by the calling thread. If this parameter is NULL, no window is associated with the timer and the nIDEvent parameter is ignored.

nIDEvent

Specifies a nonzero timer identifier. If the hWnd parameter is NULL, this parameter is ignored.

uElapse

Specifies the time-out value, in milliseconds.

lpTimerFunc

Points to the function to be notified when the time-out value elapses. For more information about the function, see TimerProc.

If lpTimerFunc is NULL, the system posts a WM_TIMER message to the application queue. The hwnd member of the message's MSG structure contains the value of the hWnd parameter.

Return Values

If the function succeeds, the return value is an integer identifying the new timer. An application can pass this value, or the string identifier, if it exists, to the KillTimer function to destroy the timer. If the function fails to create a timer, the return value is zero.

Remarks

An application can process WM_TIMER messages by including a WM_TIMER case statement in the window procedure or by specifying a TimerProc callback function when creating the timer. When you specify a TimerProc callback function, the default window procedure calls the callback function when it processes WM_TIMER. Therefore, you need to dispatch messages in the calling thread, even when you use TimerProc instead of processing WM_TIMER.

The wParam parameter of the WM_TIMER message contains the value of the nIDEvent parameter.

See Also

KillTimer, MSG, TimerProc, WM_TIMER

GetFileAttributes

The GetFileAttributes function returns attributes for a specified file or directory.

DWORD GetFileAttributes(
LPCTSTR lpFileName   // address of the name of a file or directory
);

Parameters

lpFileName

Points to a null-terminated string that specifies the name of a file or directory.

Windows NT:

There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the GetFileAttributes function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of GetFileAttributes and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with GetFileAttributesW. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps" is seen as "\\bill_g_1\hotstuff\coolapps".

Windows 95:

The lpFileName string must not exceed MAX_PATH characters. Windows 95 does not support the "\\?\" prefix.

Return Values

If the function succeeds, the return value contains the attributes of the specified file or directory.

If the function fails, the return value is 0xFFFFFFFF. To get extended error information, call GetLastError.

The attributes can be one or more of the following values:

FILE_ATTRIBUTE_ARCHIVE
The file or directory is an archive file or directory. Applications use this flag to mark files for backup or removal.

FILE_ATTRIBUTE_COMPRESSED
The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.

FILE_ATTRIBUTE_DIRECTORY
The "file or directory" is a directory.

FILE_ATTRIBUTE_HIDDEN
The file or directory is hidden. It is not included in an ordinary directory listing.

FILE_ATTRIBUTE_NORMAL
The file or directory has no other attributes set. This attribute is valid only if used alone.

FILE_ATTRIBUTE_OFFLINE
The data of the file is not immediately available. Indicates that the file data has been physically moved to offline storage.

FILE_ATTRIBUTE_READONLY
The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.

FILE_ATTRIBUTE_SYSTEM
The file or directory is part of, or is used exclusively by, the operating system.

FILE_ATTRIBUTE_TEMPORARY
The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

See Also

DeviceIOControl, FindFirstFile, FindNextFile, SetFileAttributes

GetFileAttributesEx

The GetFileAttributesEx function obtains attribute information about a specified file or directory.

This function is similar to the GetFileAttributes function. GetFileAttributes returns a set of FAT-style attribute information. GetFileAttributesEx is designed to obtain other sets of file or directory attribute information. Currently, GetFileAttributeEx obtains a set of standard attributes that is a superset of the FAT-style attribute information.

BOOL GetFileAttributesEx(
LPCTSTR lpFileName,   // pointer to string that specifies a file or directory
GET_FILEEX_INFO_LEVELS fInfoLevelId,   // value that specifies the type of attribute information to obtain
LPVOID lpFileInformation   // pointer to buffer to receive attribute information
);

Parameters

lpFileName

Pointer to a null-terminated string that specifies a file or directory.

By default, this string is limited to MAX_PATH characters. The limit is related to how the GetFileAttributesEx function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of GetFileAttributesEx and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing. This technique also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\peanuts\hotstuff\coolapps" is seen as "\\peanuts\hotstuff\coolapps".

fInfoLevelId

Specifies a GET_FILEEX_INFO_LEVELS enumeration type that gives the set of attribute information to obtain.

lpFileInformation

Pointer to a buffer that receives the attribute information. The type of attribute information stored into this buffer is determined by the value of fInfoLevelId.

Return Values

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

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

See Also

GetFileAttributes, GET_FILEEX_INFO_LEVELS, SetFileAttributes

MultiByteToWideChar

The MultiByteToWideChar function maps a character string to a wide-character (Unicode) string. The character string mapped by this function is not necessarily from a multibyte character set.

```
int MultiByteToWideChar(
UINT CodePage,          // code page
DWORD dwFlags,          // character-type options
LPCSTR lpMultiByteStr, // address of string to map
int cchMultiByte,       // number of characters in string
LPWSTR lpWideCharStr, // address of wide-character buffer
int cchWideChar         // size of buffer
);
```

Parameters

CodePage

Specifies the code page to be used to perform the conversion. This parameter can be given the value of any codepage that is installed or available in the system. The following values may be used to specify one of the system default code pages:

CP_ACP
ANSI code page

CP_MACCP
Macintosh code page

CP_OEMCP
OEM code page

dwFlags

A set of bit flags that indicate whether to translate to precomposed or composite wide characters (if a composite form exists), whether to use glyph characters in place of control characters, and how to deal with invalid characters. You can specify a combination of the following flag constants:

MB_PRECOMPOSED
Always use precomposed characters ¾ that is, characters in which a base character and a nonspacing character have a single character value. This is the default translation option. Cannot be used with MB_COMPOSITE.

MB_COMPOSITE
Always use composite characters ¾ that is, characters in which a base character and a nonspacing character have different character values. Cannot be used with MB_PRECOMPOSED.

MB_ERR_INVALID_CHARS
If the function encounters an invalid input character, it fails and GetLastError returns ERROR_NO_UNICODE_TRANSLATION.

MB_USEGLYPHCHARS
Use glyph characters instead of control characters.

A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a base/non-spacing character combination. In the character è, the e is the base character and the accent grave mark is the nonspacing character.

The function's default behavior is to translate to the precomposed form. If a precomposed form does not exist, the function attempts to translate to a composite form.

The flags MB_PRECOMPOSED and MB_COMPOSITE are mutually exclusive. The MB_USEGLYPHCHARS flag and the MB_ERR_INVALID_CHARS can be set regardless of the state of the other flags.

lpMultiByteStr

Points to the character string to be converted.

cchMultiByte

Specifies the size in bytes of the string pointed to by the lpMultiByteStr parameter. If this value is -1, the string is assumed to be null terminated and the length is calculated automatically.

lpWideCharStr

Points to a buffer that receives the translated string.

cchWideChar

Specifies the size, in wide characters, of the buffer pointed to by the lpWideCharStr parameter. If this value is zero, the function returns the required buffer size, in wide characters, and makes no use of the lpWideCharStr buffer.

Return Values

If the function succeeds, and cchWideChar is nonzero, the return value is the number of wide characters written to the buffer pointed to by lpWideCharStr.

If the function succeeds, and cchWideChar is zero, the return value is the required size, in wide characters, for a buffer that can receive the translated string.

If the function fails, the return value is zero. To get extended error information, call GetLastError. GetLastError may return one of the following error codes:

ERROR_INSUFFICIENT_BUFFER

ERROR_INVALID_FLAGS

ERROR_INVALID_PARAMETER

ERROR_NO_UNICODE_TRANSLATION

Remarks

The lpMultiByteStr and lpWideCharStr pointers must not be the same. If they are the same, the function fails, and GetLastError returns the value ERROR_INVALID_PARAMETER.

The function fails if MB_ERR_INVALID_CHARS is set and it encounters an invalid character in the source string. An invalid character is one that would translate to the default character if MB_ERR_INVALID_CHARS was not set, but is not the default character in the source string, or when a lead byte is found in a string and there is no valid trail byte for DBCS strings. When an invalid character is found, and MB_ERR_INVALID_CHARS is set, the function returns 0 and sets GetLastError with the error ERROR_NO_UNICODE_TRANSLATION.

See Also

WideCharToMultiByte

About +Sync

I have been in the scene for almost 10 years now. The first game I ever patched was Mike Ditka's Ultimate Football by Accolade, and I was hooked ever since. I struggled to learn cracking, as no texts on the subject existed at the time. I found Cyborg's Cracking Manual and studied hard. However it was not until I met an up and coming cracker named Ed!son on irc that my career really took off. I proofread his very first tutorial (the first modern text on the subject of windows cracking), and to this day I have the only copy of this text, as it was revised completely before release. I belonged to a small group (which I miss very much) known as the UGCC or Underground Coalition of Crackers for a time. I later joined Phrozen Crew for a short time, until heavy workload forced me to retire. In the summer of 1996 I stumbled accross a 'Strainer' into the Higher Cracking University. I wrote my essay, which I did not think was very good, and turned it in. That essay can be read now as +Orc's lesson C3. I recieved a great amount of my knowledge from +Orc, whose essays are wonderful and to +Fravia for collecting great works by many talented crackers. One I would like to mention specifically is Razzia, whose work has absolutely amazed me. Good work my friend. And Ed!son, if you are still out there somewhere - thank you.

GetDriveType

The GetDriveType function determines whether a disk drive is a removable, fixed, CD-ROM, RAM disk, or network drive.

```
UINT GetDriveType(
LPCTSTR lpRootPathName      // address of root path
);
```

Parameters

lpRootPathName

Points to a null-terminated string that specifies the root directory of the disk to return information about. If lpRootPathName is NULL, the function uses the root of the current directory.

Return Values

The return value specifies the type of drive. It can be one of the following values:

DRIVE_UNKNOWN
The drive type cannot be determined.

DRIVE_NO_ROOT_DIR
The root directory does not exist.

DRIVE_REMOVABLE
The disk can be removed from the drive.

DRIVE_FIXED
The disk cannot be removed from the drive.

DRIVE_REMOTE
The drive is a remote (network) drive.

DRIVE_CDROM
The drive is a CD-ROM drive.

DRIVE_RAMDISK
The drive is a RAM disk.

See Also

GetDiskFreeSpace