1 Kernel32.lib

1.1 AddAtom

The **AddAtom** function adds a character string to the local atom table and returns a unique value (an atom) identifying the string.

```
AddAtom: procedure
(
     lpString: string
);
     stdcall;
     returns( "eax" );
     external( "__imp__AddAtomA@4" );
```

Parameters

lpString

[in] Pointer to the null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings differing only in case are considered identical. The case of the first string added is preserved and returned by the **GetAtomName** function.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value in EAX is the newly created atom.

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

Remarks

The **AddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom and, in the case of a string atom, increments the string's reference count.

If *lpString* has the form "#1234", **AddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **AddAtom** returns a string atom.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.

Library: Use Kernel32.lib.

1.2 AllocConsole

The **AllocConsole** function allocates a new console for the calling process.

```
AllocConsole: procedure;
     stdcall;
     returns( "eax" ); // Zero if failure
     external( "__imp__AllocConsole@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value in EAX is nonzero.

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

Remarks

A process can be associated with only one console, so the **AllocConsole** function fails if the calling process already has a console. A process can use the **FreeConsole** function to detach itself from its current console, and then it can call **AllocConsole** to create a new console. If the calling process creates a child process, the child inherits the new console.

AllocConsole also sets up standard input, standard output, and standard error handles for the new console. The standard input handle is a handle to the console's input buffer, and the standard output and standard error handles are handles to the console's screen buffer. To retrieve these handles, use the **GetStdHandle** function.

This function is primarily used by graphics applications to create a console window. Graphics applications are initialized without a console. Console applications are normally initialized with a console, unless they are created as detached processes (by calling the **CreateProcess** function with the DETACHED_PROCESS flag).

Requirements

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

Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

1.3 AreFileApisANSI

The **AreFileApisANSI** function determines whether the file I/O functions are using the ANSI or OEM character set code page. This function is useful for 8-bit console input and output operations.

```
AreFileApisANSI: procedure;
     stdcall;
     returns( "eax" );  // Zero for OEM code page, non-zero for ANSI.
     external( "__imp__AreFileApisANSI@0" );
```

Parameters

This function has no parameters.

Return Values

If the set of file I/O functions is using the ANSI code page, the return value is nonzero.

If the set of file I/O functions is using the OEM code page, the return value is zero.

Remarks

The **SetFileApisToOEM** function causes a set of file I/O functions to use the OEM code page. The **SetFileApis-**

ToANSI function causes the same set of file I/O functions to use the ANSI code page. Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. For a discussion of these functions' usage, please refer to the Remarks sections of **SetFileApisToOEM** and **SetFileApisToANSI**.

The file I/O functions whose code page is ascertained by **AreFileApisANSI** are those functions exported by KERNEL32.DLL that accept or return a file name.

The functions **SetFileApisToOEM** and **SetFileApisToANSI** set the code page for a process, so **AreFileApisANSI** returns a value indicating the code page of an entire process.

Requirements

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

Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf.

Library: Use Kernel32.lib.

1.4 AssignProcessToJobObject

The **AssignProcessToJobObject** function associates a process with an existing job object.

```
AssignProcessToJobObject: procedure
(
     hJob:dword;
     hProcess:dword
);
     stdcall;
     returns( "eax" );
     external( "__imp__AssignProcessToJobObject@8" );
```

Parameters

hJob

[in] Handle to the job object to which the process will be associated. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_ASSIGN_PROCESS access right associated with it. For more information, see Job Object Security and Access Rights.

hProcess

[in] Handle to the process to associate with the job object. The process must not already be assigned to a job. The handle must have PROCESS_SET_QUOTA and PROCESS_TERMINATE access to the process. For more information, see Process Security and Access Rights.

Terminal Services: All processes within a job must run within the same session.

Return Values

If the function succeeds, the return value in EAX is nonzero.

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

Remarks

After you associate a process with a job object using **AssignProcessToJobObject**, the process is subject to the limits set for the job. To set limits for a job, use the **SetInformationJobObject** function.

If the job has a user-mode time limit, and the time limit has been exhausted, **AssignProcessToJobObject** fails and the specified process is terminated. If the time limit would be exceeded by associating the process, **AssignProcessToJobObject** still succeeds. However, the time limit violation will be reported. If the job has an active process limit, and the limit would be exceeded by associating this process, **AssignProcessToJobObject** fails, and the specified process is terminated.

Memory operations performed by a process associated with a job that has a memory limit are subject to the memory

limit. Memory operations performed by the process before it was associated with the job are not examined by **AssignProcessToJobObject**.

If the process is already running and the job has security limitations, **AssignProcessToJobObject** may fail. For example, if the primary token of the process contains the local administrators group, but the job object has the security limitation JOB_OBJECT_SECURITY_NO_ADMIN, the function fails. If the job has the security limitation JOB_OBJECT_SECURITY_ONLY_TOKEN, the process must be created suspended. To create a suspended process, call the **CreateProcess** function with the CREATE_SUSPENDED flag.

A process can be associated only with a single job. A process inherits limits from the job it is associated with and adds its accounting information to the job. If a process is associated with a job, all processes it creates are associated with that job by default. To create a process that is not part of the same job, call the **CreateProcess** function with the CREATE_BREAKAWAY_FROM_JOB flag.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Unsupported.

Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

1.5 BackupRead

The **BackupRead** function reads data associated with a specified file or directory into a buffer. You use this function to back up a file or directory.

```
BackupRead: procedure
(
        hFile:                  dword;
    var lpBuffer:               var;
        nNumberOfBytesToRead:   dword;
    var lpNumberOfBytesRead:    dword;
        bAbort:                 boolean;
        bProcessSecurity:       boolean;
    var lpContext:              var
);
    stdcall;
    returns( "eax" );
    external( "__imp__BackupRead@28" );
```

Parameters

hFile

[in] Handle to the file or directory being backed up. The function reads data associated with this file. You obtain this handle by calling the **CreateFile** function.

The **BackupRead** function fails if **CreateFile** was called with the flag FILE_FLAG_NO_BUFFERING. In this case, the **GetLastError** function returns the value ERROR_INVALID_PARAMETER.

lpBuffer

[out] Pointer to a buffer that the function writes data to.

nNumberOfBytesToRead

[in] Specifies the length of the buffer. The buffer size must be greater than the size of a **WIN32_STREAM_ID** structure.

lpNumberOfBytesRead

[out] Pointer to a variable that receives the number of bytes read.

If the function returns a nonzero value, and the variable pointed to by *lpNumberOfBytesRead* is zero, then all the data associated with the file handle has been read.

bAbort

[in] Indicates whether you have finished using **BackupRead** on the handle. While you are backing up the file, specify this parameter as FALSE. Once you are done using **BackupRead**, you must call **BackupRead** one more time specifying TRUE for this parameter and passing the appropriate *lpContext*. *lpContext* must be passed when *bAbort* is TRUE; all other parameters are ignored.

bProcessSecurity

[in] Indicates whether the function will restore the access-control list (ACL) data for the file or directory.

If *bProcessSecurity* is TRUE, the ACL data will be backed up.

lpContext

[out] Pointer to a variable that receives and holds a pointer to an internal data structure used by **BackupRead** to maintain context information during a backup operation.

You must set the variable pointed to by *lpContext* to NULL before the first call to **BackupRead** for the specified file or directory. The function allocates memory for the data structure, and then sets the variable to point to that structure. You must not change *lpContext* or the variable that it points to between calls to **BackupRead**.

To release the memory used by the data structure, call **BackupRead** with the *bAbort* parameter set to TRUE when the backup operation is complete.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an I/O error occurred. To get extended error information, call **GetLastError**.

Remarks

BackupRead processes all of the data pertaining to an opened object as a series of discrete byte streams. Each stream is preceded by a 32-bit aligned **WIN32_STREAM_ID** structure.

Streams must be processed in the same order in which they were written to the tape. This ordering enables applications to compare the data backed up against the data on the source device. The data returned by **BackupRead** is to be used only as input to the **BackupWrite** function. This data is returned as one large data stream divided into substreams. The substreams are separated by **WIN32_STREAM_ID** headers.

If an error occurs while **BackupRead** is reading, the calling process can skip the bad data by calling the **BackupSeek** function.

Requirements

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

Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

1.6 BackupSeek

The **BackupSeek** function seeks forward in a data stream initially accessed by using the **BackupRead** or **BackupWrite** function.

```
BackupSeek: procedure
(
        hFile:              dword;
        dwLowBytesToSeek:   dword;
        dwHighBytesToSeek:  dword;
```

```
        var lpdwLowByteSeeked:  dword;
        var lpdwHighByteSeeked: dword;
        var lpContext:          dword
    );
        stdcall;
        returns( "eax" );
        external( "__imp__BackupSeek@24" );
```

Parameters

hFile

[in] Handle to the file or directory being backed up. This handle is created by using the **CreateFile** function.

dwLowBytesToSeek

[in] Specifies the low-order bits of the number of bytes to seek.

dwHighBytesToSeek

[in] Specifies the high-order bits of the number of bytes to seek.

lpdwLowByteSeeked

[out] Pointer to a variable that receives the low-order bits of the number of bytes the function actually seeks.

lpdwHighByteSeeked

[out] Pointer to a variable that receives the high-order bits of the number of bytes the function actually seeks.

lpContext

[in] Pointer to an internal data structure used by the function. This structure must be the same structure that was initialized by the **BackupRead** function. An application must not touch the contents of this structure.

Return Values

If the function could seek the requested amount, the function returns a nonzero value.

If the function could not seek the requested amount, the function returns zero. To get extended error information, call **GetLastError**.

Remarks

Applications use the **BackUpSeek** function to skip portions of a data stream that cause errors. This function does not seek across stream headers. If an application attempts to seek past the end of a substream, the function fails, the *lpdwLowByteSeeked* and *lpdwHighByteSeeked* parameters indicate the actual number of bytes the function seeks, and the file position is placed at the start of the next stream header.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, BackupRead, BackupWrite, CreateFile

---

1.7 BackupWrite

The **BackupWrite** function writes a stream of data from a buffer to a specified file or directory. The data must be divided into substreams separated by **WIN32_STREAM_ID** structures. You use this function to restore a file or directory that has been backed up.

```
BackupWrite: procedure
```

```
(
        hFile:                  dword;
    var lpBuffer:               var;
        nNumberOfBytesToWrite:  dword;
    var lpNumberOfBytesWritten: dword;
        bAbort:                 boolean;
        bProcessSecurity:       boolean;
    var lpContext:              var
);
    stdcall;
    returns( "eax" );
    external( "__imp__BackupWrite@28" );
```

Parameters

hFile

[in] Handle to the file or directory being restored. The function writes data to this file. You obtain this handle by calling the **CreateFile** function.

The **BackupWrite** function fails if **CreateFile** was called with the flag FILE_FLAG_NO_BUFFERING. In this case, the **GetLastError** function returns the value ERROR_INVALID_PARAMETER.

lpBuffer

[in] Pointer to a buffer that the function writes data from.

nNumberOfBytesToWrite

[in] Specifies the length of the buffer. The buffer size must be greater than the size of a **WIN32_STREAM_ID** structure.

lpNumberOfBytesWritten

[out] Pointer to a variable that receives the number of bytes written.

bAbort

[in] Indicates whether you have finished using **BackupWrite** on the handle. While you are backing up the file, specify this parameter as FALSE. After you are done using **BackupWrite**, you must call **BackupWrite** one more time specifying TRUE for this parameter and passing the appropriate *lpContext*. *lpContext* must be passed when *bAbort* is TRUE; all other parameters are ignored.

bProcessSecurity

[in] Specifies whether the function will restore the access-control list (ACL) data for the file or directory.

If *bProcessSecurity* is TRUE, you need to have specified WRITE_OWNER and WRITE_DAC access when opening the file or directory handle. If the handle does not have those access rights, the operating system denies access to the ACL data, and ACL data restoration will not occur.

lpContext

[out] Pointer to a variable that receives a pointer to an internal data structure used by **BackupWrite** to maintain context information during a restore operation.

You must set the variable pointed to by *lpContext* to NULL before the first call to **BackupWrite** for the specified file or directory. The function allocates memory for the data structure, and then sets the variable to point to that structure. You must not change *lpContext* or the variable that it points to between calls to **BackupWrite**.

To release the memory used by the data structure, call **BackupWrite** with the *bAbort* parameter set to TRUE when the restore operation is complete.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an I/O error occurred. To get extended error information, call **GetLastError**.

Remarks

The BACKUP_LINK stream type lets you restore files with hard links.

Data obtained by the **BackupRead** function should only be used as input to the **BackupWrite** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, BackupRead, BackupSeek, CreateFile, WIN32_STREAM_ID

1.8 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.

```
Beep: procedure
(
    dwFreq:     dword;
    dwDuration: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__Beep@8" );
```

Parameters

dwFreq

Windows NT/ 2000: [in] Specifies the frequency, in hertz, of the sound. This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).

dwDuration

Windows NT/ 2000: [in] Specifies the duration, in milliseconds, of the sound.

Return Values

If the function succeeds, the return value in EAX 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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions, MessageBeep

1.9 BeginUpdateResource

The **BeginUpdateResource** function returns a handle that can be used by the **UpdateResource** function to add, delete, or replace resources in an executable file.

```
BeginUpdateResource: procedure
(
    filename: string;
    bDeleteExistingResources: boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__BeginUpdateResourceA@8" );
```

Parameters

pFileName

[in] Pointer to a null-terminated string that specifies the executable file in which to update resources. An application must be able to obtain write access to this file; it cannot be currently executing. If *pFileName* does not specify a full path, the system searches for the file in the current directory.

bDeleteExistingResources

[in] Specifies whether to delete the *pFileName* parameter's existing resources. If this parameter is TRUE, existing resources are deleted and the updated executable file includes only resources added with the **UpdateResource** function. If this parameter is FALSE, the updated executable file includes existing resources unless they are explicitly deleted or replaced by using **UpdateResource**.

Return Values

If the function succeeds, the return value in EAX is a handle that can be used by the **UpdateResource** and **EndUpdateResource** functions. The return value is NULL if the specified file is not an executable file, the executable file is already loaded, the file does not exist, or the file cannot be opened for writing. To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, EndUpdateResource, UpdateResource

1.10 BuildCommDCB

The **BuildCommDCB** function fills a specified **DCB** structure with values specified in a device-control string. The device-control string uses the syntax of the **mode** command.

```
BuildCommDCB: procedure
(
        lpDef:  string;
    var lpDCB:  DCB
```

```
    );
        stdcall;
        returns( "eax" );
        external( "__imp__BuildCommDCBA@8" );
```

Parameters

lpDef

[in] Pointer to a null-terminated string that specifies device-control information. The string must have the same form as the **mode** command's command-line arguments. For example, the following string specifies a baud rate of 1200, no parity, 8 data bits, and 1 stop bit:

```
baud=1200 parity=N data=8 stop=1
```

The device name is ignored if it is included in the string, but it must specify a valid device, as follows:

```
COM1: baud=1200 parity=N data=8 stop=1
```

For further information on **mode** command syntax, refer to the end-user documentation for your operating system.

lpDCB

[out] Pointer to a **DCB** structure that receives the information.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **BuildCommDCB** function adjusts only those members of the **DCB** structure that are specifically affected by the *lpDef* parameter, with the following exceptions:

If the specified baud rate is 110, the function sets the stop bits to 2 to remain compatible with the system's **mode** command.

By default, **BuildCommDCB** disables XON/XOFF and hardware flow control. To enable flow control, you must explicitly set the appropriate members of the **DCB** structure.

The **BuildCommDCB** function only fills in the members of the **DCB** structure. To apply these settings to a serial port, use the **SetCommState** function.

There are older and newer forms of the **mode** command syntax. The **BuildCommDCB** function supports both forms. However, you cannot mix the two forms together.

The newer form of the **mode** command syntax lets you explicitly set the values of the flow control members of the **DCB** structure. If you use an older form of the **mode** syntax, the **BuildCommDCB** function sets the flow control members of the **DCB** structure, as follows:

For a string such as **96,n,8,1** or any other older-form **mode** string that doesn't end with an **x** or a **p**:

fInX, **fOutX**, **fOutXDsrFlow**, and **fOutXCtsFlow** are all set to FALSE

fDtrControl is set to DTR_CONTROL_ENABLE

fRtsControl is set to RTS_CONTROL_ENABLE

For a string such as **96,n,8,1,x** or any other older-form **mode** string that ends with an **x**:

fInX and **fOutX** are both set to TRUE

fOutXDsrFlow and **fOutXCtsFlow** are both set to FALSE

fDtrControl is set to DTR_CONTROL_ENABLE

fRtsControl is set to RTS_CONTROL_ENABLE

For a string such as **96,n,8,1,p** or any other older-form **mode** string that ends with a **p**:

fInX and **fOutX** are both set to FALSE

fOutXDsrFlow and **fOutXCtsFlow** are both set to TRUE

fDtrControl is set to DTR_CONTROL_HANDSHAKE

fRtsControl is set to RTS_CONTROL_HANDSHAKE

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, DCB, SetCommState

---

1.11 BuildCommDCBAndTimeouts

The **BuildCommDCBAndTimeouts** function translates a device-definition string into appropriate device-control block codes and places them into a device control block. The function can also set up time-out values, including the possibility of no time-outs, for a device; the function's behavior in this regard varies based on the contents of the device-definition string.

```
BuildCommDCBAndTimeouts: procedure
(
        lpDef:          string;
    var lpDCB:          DCB;
    var lpCommTimeouts: COMMTIMEOUTS
);
    stdcall;
    returns( "eax" );
    external( "__imp__BuildCommDCBAndTimeoutsA@12" );
```

Parameters

lpDef

[in] Pointer to a null-terminated string that specifies device-control information for the device. The function takes this string, parses it, and then sets appropriate values in the **DCB** structure pointed to by *lpDCB*.

lpDCB

[out] Pointer to a **DCB** structure that receives information from the device-control information string pointed to by *lpDef*. This **DCB** structure defines the control settings for a communications device.

lpCommTimeouts

[in] Pointer to a **COMMTIMEOUTS** structure that the function can use to set device time-out values.

The **BuildCommDCBAndTimeouts** function modifies its time-out setting behavior based on the presence or absence of a "TO=xxx" substring in the string specified by *lpDef*:

If that string contains the substring "TO=ON", the function sets up total read and write time-out values for the device based on the time-out structure pointed to by *lpCommTimeouts*.

If that string contains the substring "TO=OFF", the function sets up the device with no time-outs.

If that string contains neither the "TO=ON" substring nor the "TO=OFF" substring, the function ignores the time-out structure pointed to by *lpCommTimeouts*. The time-out structure will not be accessed.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, BuildCommDCB, COMMTIMEOUTS, DCB, GetCommTimeouts, SetCommTimeouts

---

1.12 CallNamedPipe

The **CallNamedPipe** function connects to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe.

```
CallNamedPipe: procedure
(
        lpNamedPipeName:    string;
    var lpInBuffer:         var;
        nInBufferSize:      dword;
    var lpOutBuffer:        var;
        nOutBufferSize:     dword;
    var lpBytesRead:        dword;
        nTimeOut:           dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CallNamedPipeA@28" );
```

Parameters

lpNamedPipeName

[in] Pointer to a null-terminated string specifying the pipe name.

lpInBuffer

[in] Pointer to the buffer containing the data written to the pipe.

nInBufferSize

[in] Specifies the size, in bytes, of the write buffer.

lpOutBuffer

[out] Pointer to the buffer that receives the data read from the pipe.

nOutBufferSize

[in] Specifies the size, in bytes, of the read buffer.

lpBytesRead

[out] Pointer to a variable that receives the number of bytes read from the pipe.

nTimeOut

[in] Specifies the number of milliseconds to wait for the named pipe to be available. In addition to numeric values, the following special values can be specified.

Value	Meaning
NMPWAIT_NOWAIT	Does not wait for the named pipe. If the named pipe is not available, the function returns an error.
NMPWAIT_WAIT_FOREVER	Waits indefinitely.
NMPWAIT_USE_DEFAULT_WAIT	Uses the default time-out specified in a call to the **CreateNamedPipe** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Calling **CallNamedPipe** is equivalent to calling the **CreateFile** (or **WaitNamedPipe**, if **CreateFile** cannot open the pipe immediately), **TransactNamedPipe**, and **CloseHandle** functions. **CreateFile** is called with an access flag of GENERIC_READ | GENERIC_WRITE, an inherit handle flag of FALSE, and a share mode of zero (indicating no sharing of this pipe instance).

If the message written to the pipe by the server process is longer than *nOutBufferSize*, **CallNamedPipe** returns FALSE, and **GetLastError** returns ERROR_MORE_DATA. The remainder of the message is discarded, because **CallNamedPipe** closes the handle to the pipe before returning.

CallNamedPipe fails if the pipe is a byte-type pipe.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CloseHandle, CreateFile, CreateNamedPipe, TransactNamedPipe, WaitNamedPipe

---

1.13 CancelWaitableTimer

The **CancelWaitableTimer** function sets the specified waitable timer to the inactive state.

```
CancelWaitableTimer: procedure
(
    hTimer:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CancelWaitableTimer@4" );
```

Parameters

hTimer

[in] Handle to the timer object. The **CreateWaitableTimer** or **OpenWaitableTimer** function returns this handle.

Return Values

If the function succeeds, the return value in EAX is nonzero.

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

Remarks

The **CancelWaitableTimer** function does not change the signaled state of the timer. It stops the timer before it can be set to the signaled state and cancels outstanding APCs. Therefore, threads performing a wait operation on the timer remain waiting until they time out or the timer is reactivated and its state is set to signaled. If the timer is already in the signaled state, it remains in that state.

To reactivate the timer, call the **SetWaitableTimer** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateWaitableTimer, OpenWaitable-Timer, SetWaitableTimer

---

1.14 CloseHandle

The **CloseHandle** function closes an open object handle.

```
CloseHandle: procedure
(
    handle:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CloseHandle@4" );
```

Parameters

hObject

[in/out] Handle to an open object.

Return Values

If the function succeeds, the return value is nonzero.

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

Windows NT/2000: Closing an invalid handle raises an exception when the application is running under a debugger. This includes closing a handle twice, and using **CloseHandle** on a handle returned by the **FindFirstFile** function.

Remarks

The **CloseHandle** function closes handles to the following objects:

Access token

Communications device

Console input

Console screen buffer

Event

File

File mapping

Job

Mailslot

Mutex

Named pipe

Process

Semaphore

Socket

Thread

CloseHandle invalidates the specified object handle, decrements the object's handle count, and performs object retention checks. After the last handle to an object is closed, the object is removed from the system.

Closing a thread handle does not terminate the associated thread. To remove a thread object, you must terminate the thread, then close all handles to the thread.

Use **CloseHandle** to close handles returned by calls to the **`CreateFile`** function. Use **`FindClose`** to close handles returned by calls to **`FindFirstFile`**.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.

See Also

Handles and Objects Overview, Handle and Object Functions, CreateFile, DeleteFile, FindClose, FindFirstFile

---

1.15 CommConfigDialog

The **CommConfigDialog** function displays a driver-supplied configuration dialog box.

```
CommConfigDialog: procedure
(
        lpszName:   string;
        hWnd:       dword;
    var lpCC:       COMMCONFIG
);
    stdcall;
    returns( "eax" );
    external( "__imp__CommConfigDialogA@12" );
```

Parameters

lpszName

[in] Pointer to a null-terminated string specifying the name of the device for which a dialog box should be displayed.

hWnd

[in] Handle to the window that owns the dialog box. This parameter can be any valid window handle, or it should be NULL if the dialog box is to have no owner.

lpCC

[in/out] Pointer to a **COMMCONFIG** structure. This structure contains initial settings for the dialog box before the call, and changed values after the call.

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 **CommConfigDialog** function requires a dynamic-link library (DLL) provided by the communications hardware vendor.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.
See Also

Communications Overview, Communication Functions, COMMCONFIG

1.16 CompareFileTime

The **CompareFileTime** function compares two 64-bit file times.

```
CompareFileTime: procedure
(
    var lpfileTime1:    FILETIME;
    var lpfileTime2:    FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__CompareFileTime@8" );
```

Parameters

lpFileTime1

[in] Pointer to a **FILETIME** structure that specifies the first 64-bit file time.

lpFileTime2

[in] Pointer to a **FILETIME** structure that specifies the second 64-bit file time.

Return Values

The return value is one of the following values.

Value	Meaning
–1	First file time is less than second file time.

0	First file time is equal to second file time.
1	First file time is greater than second file time.

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetFileTime, FILETIME

1.17 CompareString

The **CompareString** function compares two character strings, using the specified locale.

```
CompareString: procedure
(
    Locale:     LCID;
    dwCmpFlags: dword;
    lpString1:  string;
    cchCount1:  uns32;
    lpString2:  string;
    cchCount2:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CompareStringA@24" );
```

Parameters

Locale

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

Value	Meaning
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

[in] 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.

Value	Meaning
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

[in] Pointer to the first string to be compared.

cchCount1

[in] Specifies the number of **TCHARs** in the string pointed to by the *lpString1* parameter. This refers to bytes for ANSI versions of the function or characters for Unicode versions. The count does not include the null-terminator. If this parameter is –1, the string is assumed to be null terminated and the length is calculated automatically.

lpString2

[in] Pointer to the second string to be compared.

cchCount2

[in] Specifies the number of **TCHARs** in the string pointed to by the *lpString2* parameter. The count does not include the null-terminator. 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.

Value	Meaning
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 CSTR_EQUAL, 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 **Locales**.

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.

For DBCS locales, the flag NORM_IGNORECASE has an effect on all the wide (two-byte) characters as well as the narrow (one-byte) characters. This includes the wide Greek and Cyrillic characters.

In Chinese Simplified, the sorting order used to compare the strings is based on the following sequence: symbols, digit numbers, English letters, and Chinese Simplified characters. The characters within each group sort in character-code order.

In Chinese Traditional, the sorting order used to compare strings is based on the number of strokes in the charaters. Symbols, digit numbers, and English characters are considered to have zero strokes. The sort sequence is symbols, digit numbers, English letters, and Chinese Traditional characters. The characters within each stroke-number group sort in character-code order.

In Japanese, the sorting order used to compare the strings is based on the Japanese 50-on sorting sequence. The Kanji ideographic characters sort in character-code order.

In Japanese, the flag NORM-IGNORENONSPACE has an effect on the daku-on, handaku-on, chou-on, you-on, and soku-on modifiers, and on the repeat kana/kanji characters.

In Korean, the sort order is based on the sequence: symbols, digit numbers, Jaso and Hangeul, Hanja, and English. Within the Jaso-Hangeul group, each Jaso character is followed by the Hangeuls that start with that Jaso. Hanja characters are sorted in Hangeul pronunciation order. Where multiple Hanja have the same Hangeul pronunciation, they are sorted in character-code order.

The NORM_IGNORENONSPACE flag only has an effect for the locales in which accented characters are sorted in a second pass from main characters. All characters in the string are first compared without regard to accents and (if the strings are equal) a second pass over the strings is performed to compare accents. In this case, this flag causes the second pass to not be performed. For locales that sort accented characters in the first pass, this flag has no effect.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnls.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, FoldString, GetSystemDefaultLCID, GetUserDefaultLCID, LCMapString, lstrcmp, lstrcmpi, MAKELCID

1.18 ConnectNamedPipe

The **ConnectNamedPipe** function enables a named pipe server process to wait for a client process to connect to an instance of a named pipe. A client process connects by calling either the **CreateFile** or **CallNamedPipe** function.

```
ConnectNamedPipe: procedure
(
        hNamedPipe:     dword;
    var lpOverlapped:   OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__ConnectNamedPipe@8" );
```

Parameters

hNamedPipe

[in] Handle to the server end of a named pipe instance. This handle is returned by the **CreateNamedPipe** function.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure.

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

If *hNamedPipe* was created with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the **OVERLAPPED** structure must contain a handle to a manual-reset event object (which the server can create by using the **CreateEvent** function).

If *hNamedPipe* was not opened with FILE_FLAG_OVERLAPPED, the function does not return until a client is connected or an error occurs. Successful synchronous operations result in the function returning a nonzero value if a client connects after the function is called.

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 a client connects before the function is called, the function returns zero and **GetLastError** returns

ERROR_PIPE_CONNECTED. This can happen if a client connects in the interval between the call to **CreateNamedPipe** and the call to **ConnectNamedPipe**. In this situation, there is a good connection between client and server, even though the function returns zero.

Remarks

A named pipe server process can use **ConnectNamedPipe** with a newly created pipe instance. It can also be used with an instance that was previously connected to another client process; in this case, the server process must first call the **DisconnectNamedPipe** function to disconnect the handle from the previous client before the handle can be reconnected to a new client. Otherwise, **ConnectNamedPipe** returns zero, and **GetLastError** returns ERROR_NO_DATA if the previous client has closed its handle or ERROR_PIPE_CONNECTED if it has not closed its handle.

The behavior of **ConnectNamedPipe** depends on two conditions: whether the pipe handle's wait mode is set to blocking or nonblocking and whether the function is set to execute synchronously or in overlapped mode. A server initially specifies a pipe handle's wait mode in the **CreateNamedPipe** function, and it can be changed by using the **SetNamedPipeHandleState** function.

The server process can use any of the wait functions or **SleepEx** to determine when the state of the event object is signaled, and it can then use the **GetOverlappedResult** function to determine the results of the **ConnectNamedPipe** operation.

If the specified pipe handle is in nonblocking mode, **ConnectNamedPipe** always returns immediately. In nonblocking mode, **ConnectNamedPipe** returns a nonzero value the first time it is called for a pipe instance that is disconnected from a previous client. This indicates that the pipe is now available to be connected to a new client process. In all other situations when the pipe handle is in nonblocking mode, **ConnectNamedPipe** returns zero. In these situations, **GetLastError** returns ERROR_PIPE_LISTENING if no client is connected, ERROR_PIPE_CONNECTED if a client is connected, and ERROR_NO_DATA if a previous client has closed its pipe handle but the server has not disconnected. Note that a good connection between client and server exists only after the ERROR_PIPE_CONNECTED error is received.

Note Nonblocking mode is supported for compatibility with Microsoft LAN Manager version 2.0, and it should not be used to achieve asynchronous input and output (I/O) with named pipes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CallNamedPipe, CreateEvent, CreateFile, CreateNamedPipe, DisconnectNamedPipe, GetOverlappedResult, SetNamedPipeHandleState, SleepEx, OVERLAPPED

1.19 ContinueDebugEvent

The **ContinueDebugEvent** function enables a debugger to continue a thread that previously reported a debugging event.

```
ContinueDebugEvent: procedure
(
    dwProcessID:        dword;
    dwThreadID:         dword;
    dwContinueStatus:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ContinueDebugEvent@12" );
```

Parameters

dwProcessId

[in] Handle to the process to continue.

dwThreadId

[in] Handle to the thread to continue. The combination of process identifier and thread identifier must identify a thread that has previously reported a debugging event.

dwContinueStatus

[in] Specifies how to continue the thread that reported the debugging event.

If the DBG_CONTINUE flag is specified for this parameter and the thread specified by the *dwThreadId* parameter previously reported an EXCEPTION_DEBUG_EVENT debugging event, the function stops all exception processing and continues the thread. For any other debugging event, this flag simply continues the thread.

If the DBG_EXCEPTION_NOT_HANDLED flag is specified for this parameter and the thread specified by *dwThreadId* previously reported an EXCEPTION_DEBUG_EVENT debugging event, the function continues exception processing. If this is a first-chance exception event, the search and dispatch logic of the structured exception handler is used; otherwise, the process is terminated. For any other debugging event, this flag simply continues the thread.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Only the thread that created *dwProcessId* with the **CreateProcess** function can call **ContinueDebugEvent**.

After the **ContinueDebugEvent** function succeeds, the specified thread continues. Depending on the debugging event previously reported by the thread, different actions occur. If the continued thread previously reported an EXIT_THREAD_DEBUG_EVENT debugging event, **ContinueDebugEvent** closes the handle the debugger has to the thread. If the continued thread previously reported an EXIT_PROCESS_DEBUG_EVENT debugging event, **ContinueDebugEvent** closes the handles the debugger has to the process and to the thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, CreateProcess

1.20 ConvertDefaultLocale

The **ConvertDefaultLocale** function converts a default locale value to an actual locale identifier.

```
ConvertDefaultLocale: procedure
(
    Local:  LCID
);
    stdcall;
    returns( "eax" );
    external( "__imp__ConvertDefaultLocale@4" );
```

Parameters

Locale

[in/out] Default locale value that the function converts to a locale identifier (LCID). The following are the default locale values.

Value	Description
LOCALE_SYSTEM_DEFAULT	The system's default locale.
LOCALE_USER_DEFAULT	The current user's default locale.
A sublanguage-neutral locale	A locale identifier constructed by calling **MAKELCID** with a primary language identifier, such as LANG_ENGLISH, and the SUBLANG_NEUTRAL sublanguage identifier.

Return Values

If the function succeeds, the return value is the appropriate LCID.

If the function fails, the return value is the *Locale* parameter. The function fails when *Locale* is not one of the default locale values listed above.

Remarks

A call to **ConvertDefaultLocale**(LOCALE_SYSTEM_DEFAULT) is equivalent to a call to **GetSystemDefaultLCID**. A call to **ConvertDefaultLocale**(LOCALE_USER_DEFAULT) is equivalent to a call to **GetUserDefaultLCID**.

For more information, see Locales and Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetSystemDefaultLCID, GetUserDefaultLCID

1.21 ConvertThreadToFiber

The **ConvertTheadToFiber** function converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

```
ConvertThreadToFiber: procedure
(
    var lpParameter: var;
);
    stdcall;
    returns( "eax" );
    external( "__imp__ConvertThreadToFiber@4" );
```

Parameters

lpParameter

[in] Specifies a single variable that is passed to the fiber. The fiber can retrieve this value by using the **GetFiber-Data** macro.

Return Values

If the function succeeds, the return value is the address of the fiber.

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

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call **ConvertThreadToFiber** to create an area in which to save fiber state information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetFiberData

---

1.22 CopyFile

The **CopyFile** function copies an existing file to a new file.

The **CopyFileEx** function provides two additional capabilities. **CopyFileEx** can call a specified callback function each time a portion of the copy operation is completed, and **CopyFileEx** can be canceled during the copy operation.

```
CopyFile: procedure
(
    lpExistingFileName: string;
    lpNewFileName:      string;
    bFailIfExists:      boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__CopyFileA@12" );
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that specifies the name of an existing file.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpNewFileName

[in] Pointer to a null-terminated string that specifies the name of the new file.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

bFailIfExists

[in] Specifies how this operation is to proceed if a file of the same name as that specified by *lpNewFileName* already exists. If this parameter is TRUE and the new file already exists, the function fails. If this parameter is FALSE and the new file already exists, the function overwrites the existing file and succeeds.

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

Security attributes for the existing file are not copied to the new file.

File attributes for the existing file are copied to the new file. For example, if an existing file has the FILE_ATTRIBUTE_READONLY file attribute, a copy created through a call to **CopyFile** will also have the FILE_ATTRIBUTE_READONLY file attribute.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CopyFileEx, CreateFile, MoveFile

---

1.23 CopyFileEx

The **CopyFileEx** function copies an existing file to a new file. This function preserves extended attributes, OLE structured storage, NTFS alternate data streams, and file attributes. Security attributes for the existing file are not copied to the new file.

```
CopyFileEx: procedure
(
        lpExistingFileName: string;
        lpNewFileName:      string;
    var lpProgressRoutine:  PROGRESS_ROUTINE;
    var lpData:             var;
    var pbCancel:           boolean;
        dwCopyFlags:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CopyFileExA@24" );
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that specifies the name of an existing file.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpNewFileName

[in] Pointer to a null-terminated string that specifies the name of the new file.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpProgressRoutine

[in] Specifies the address of a callback function of type LPPROGRESS_ROUTINE that is called each time another portion of the file has been copied. This parameter can be NULL. For more information on the progress callback function, see CopyProgressRoutine.

lpData

[in] Specifies an argument to be passed to the callback function. This parameter can be NULL.

pbCancel

[in] Pointer to a Boolean variable that can be used to cancel the operation. If this flag is set to TRUE during the copy operation, the operation is canceled.

dwCopyFlags

[in] Specifies how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning
COPY_FILE_FAIL_IF_EXISTS	The copy operation fails immediately if the target file already exists.
COPY_FILE_RESTARTABLE	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for *lpExistingFileName* and *lpNew-FileName* as those used in the call that failed.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.
See Also

File I/O Overview, File I/O Functions, CreateFile, CopyFile, CopyProgressRoutine, MoveFile

1.24 CreateConsoleScreenBuffer

The **CreateConsoleScreenBuffer** function creates a console screen buffer.

```
CreateConsoleScreenBuffer: procedure
(
        dwDesiredAccess:        dword;
        dwShareMode:            dword;
    var lpSecurityAttributes:   Security_Attributes;
```

```
        dwFlags:                dword;
        lpScreenBufferData:     dword  // Should be NULL.
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateConsoleScreenBuffer@20" );
```

Parameters

dwDesiredAccess

[in] Specifies the desired access to the console screen buffer. This parameter can be one or more of the following values.

Value	Meaning
GENERIC_READ	Requests read access to the console screen buffer, enabling the process to read data from the buffer.
GENERIC_WRITE	Requests write access to the console screen buffer, enabling the process to write data to the buffer.

dwShareMode

[in] Specifies how this console screen buffer can be shared. This parameter can be zero, indicating that the buffer cannot be shared, or it can be one or more of the following values.

Value	Meaning
FILE_SHARE_READ	Other open operations can be performed on the console screen buffer for read access.
FILE_SHARE_WRITE	Other open operations can be performed on the console screen buffer for write access.

lpSecurityAttributes

[in] 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/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new console screen buffer. If *lpSecurityAttributes* is NULL, the console screen buffer gets a default security descriptor.

dwFlags

[in] Specifies the type of console screen buffer to create. The only currently supported screen buffer type is CONSOLE_TEXTMODE_BUFFER.

lpScreenBufferData

[in] Reserved; should be NULL.

Return Values

If the function succeeds, the return value is a handle to the new console screen buffer.

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

Remarks

A console can have multiple screen buffers but only one active screen buffer. Inactive screen buffers can be accessed for reading and writing, but only the active screen buffer is displayed. To make the new screen buffer the active screen buffer, use the **SetConsoleActiveScreenBuffer** function.

The calling process can use the returned handle in any function that requires a handle to a console screen buffer, subject to the limitations of access specified by the *dwDesiredAccess* parameter.

The calling process can use the **DuplicateHandle** function to create a duplicate screen buffer handle that has different access or inheritability from the original handle. However, **DuplicateHandle** cannot be used to create a duplicate that is valid for a different process (except through inheritance).

To close the screen buffer handle, use the **CloseHandle** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CloseHandle, DuplicateHandle, GetConsoleScreenBufferInfo, SECURITY_ATTRIBUTES, SetConsoleActiveScreenBuffer, SetConsoleScreenBufferSize

---

1.25 CreateDirectory

The **CreateDirectory** function creates a new directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory.

To specify a template directory, use the **CreateDirectoryEx** function.

```
CreateDirectory: procedure
(
    lpPathName:             string;
    lpSecurityAttributes:   dword   // Should be NULL
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateDirectoryA@8" );
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to be created.

There is a default string size limit for paths of 248 characters. This limit is related to how the **CreateDirectory** function parses paths.

Windows NT/2000: To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

lpSecurityAttributes

Windows NT/2000: [in] Pointer to a **SECURITY_ATTRIBUTES** structure. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is NULL, the directory gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect.

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

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

Windows NT/2000: An application can obtain a handle to a directory by calling **CreateFile** with the FILE_FLAG_BACKUP_SEMANTICS flag set. For a code example, see **`CreateFile`**.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateDirectoryEx, CreateFile, RemoveDirectory, SECURITY_ATTRIBUTES

1.26 CreateDirectoryEx

The **CreateDirectoryEx** function creates a new directory with the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

```
CreateDirectoryEx: procedure
(
    lpTemplateDirectory:    string;
    lpNewDirectory:         string;
    lpSecurityAttributes:   dword   // Should be NULL
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateDirectoryExA@12" );
```

Parameters

lpTemplateDirectory

[in] Pointer to a null-terminated string that specifies the path of the directory to use as a template when creating the new directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpNewDirectory

[in] Pointer to a null-terminated string that specifies the path of the directory to be created.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpSecurityAttributes

Windows NT/2000: [in] Pointer to a **SECURITY_ATTRIBUTES** structure. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is NULL, the directory gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect.

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 **CreateDirectoryEx** function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when dealing with Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

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

Windows NT/2000: You can obtain a handle to a directory by calling the **CreateFile** function with the FILE_FLAG_BACKUP_SEMANTICS flag set. See **CreateFile** for a code example.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateDirectory, CreateFile, RemoveDirectory, SECURITY_ATTRIBUTES

1.27 CreateEvent

The **CreateEvent** function creates or opens a named or unnamed event object.

```
CreateEvent: procedure
(
    lpEventAttributes:  dword;   // Should be NULL
    bManualReset:       boolean;
    bInitialState:      boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateEventA@16" );
```

Parameters

lpEventAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new event. If *lpEventAttributes* is NULL, the event gets a default security descriptor.

bManualReset

[in] Specifies whether a manual-reset or auto-reset event object is created. If TRUE, then you must use the **ResetEvent** function to manually reset the state to nonsignaled. If FALSE, the system automatically resets the state to nonsignaled after a single waiting thread has been released.

bInitialState

[in] Specifies the initial state of the event object. If TRUE, the initial state is signaled; otherwise, it is nonsignaled.

lpName

[in] Pointer to a null-terminated string specifying the name of the event object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named event object, this function requests EVENT_ALL_ACCESS access to the existing object. In this case, the *bManualReset* and *bInitialState* parameters are ignored because they have already been set by the creating process. If the *lpEventAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

If *lpName* is NULL, the event object is created without a name.

If *lpName* matches the name of an existing semaphore, mutex, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS.

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

Remarks

The handle returned by **CreateEvent** has EVENT_ALL_ACCESS access to the new event object and can be used in any function that requires a handle to an event object.

Any thread of the calling process can specify the event-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The initial state of the event object is specified by the *bInitialState* parameter. Use the **SetEvent** function to set the state of an event object to signaled. Use the **ResetEvent** function to reset the state of an event object to nonsignaled.

When the state of a manual-reset event object is signaled, it remains signaled until it is explicitly reset to nonsignaled by the **ResetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object, can be released while the object's state is signaled.

When the state of an auto-reset event object is signaled, it remains signaled until a single waiting thread is released; the system then automatically resets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled.

Multiple processes can have handles of the same event object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

A child process created by the **CreateProcess** function can inherit a handle to an event object if the *lpEventAttributes* parameter of **CreateEvent** enabled inheritance.
A process can specify the event-object handle in a call to the **DuplicateHandle** function to create a duplicate handle that can be used by another process.
A process can specify the name of an event object in a call to the **OpenEvent** or **CreateEvent** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

Example

For an example that uses **CreateEvent**, see Using Event Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateProcess, DuplicateHandle, OpenEvent, ResetEvent, SECURITY_ATTRIBUTES, SetEvent, Object Names

---

1.28 CreateFiber

The **CreateFiber** function allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

```
CreateFiber: procedure
(
        dwStackSize:    dword;
    var lpStartAddress: FIBER_START_ROUTINE;
        lpParameter:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateFiber@12" );
```

Parameters

dwStackSize

[in] Specifies the size, in bytes, of the stack for the new fiber. If zero is specified, the stack size defaults to the same size as that of the main thread. The function fails if it cannot commit *dwStackSize* bytes. Note that the system increases the stack size dynamically, if necessary. For more information, see Thread Stack Size.

lpStartAddress

[in] Pointer to the application-defined function of type LPFIBER_START_ROUTINE to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the **SwitchToFiber** function with this address. For more information of the fiber callback function, see **FiberProc**.

lpParameter

[in] Specifies a single argument that is passed to the fiber. This value can be retrieved by the fiber using the **GetFiberData** macro.

Return Values

If the function succeeds, the return value is the address of the fiber.

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

Remarks

Before a thread can schedule a fiber using the **SwitchToFiber** function, it must call the **ConvertThreadToFiber** function so there is a fiber associated with the thread.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, ConvertThreadToFiber, FiberProc, GetFiberData, SwitchToFiber

---

1.29 CreateFile

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

Consoles

Communications resources

Directories (open only)

Disk devices (Windows NT/2000 only)

Files

Mailslots

Pipes

```
CreateFile: procedure
(
    lpFileName:             string;
    dwDesiredAccess:        dword;
    dwShareMode:            dword;
    lpSecurityAttributes:   dword;  // Should be NULL
    dwCreationDisposition:  dword;
    dwFlagsAndAttributes:   dword;
    hTemplateFile:          dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateFileA@28" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the object to create or open.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To

extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

dwDesiredAccess

[in] 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.

Value	Meaning
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.

In addition, you can specify the following access flags.

Value	Documented
DELETE	Standard Access Rights
READ_CONTROL	Standard Access Rights
WRITE_DAC	Standard Access Rights
WRITE_OWNER	Standard Access Rights
SYNCHRONIZE	Standard Access Rights
STANDARD_RIGHTS_REQUIRED	Standard Access Rights
STANDARD_RIGHTS_READ	Standard Access Rights
STANDARD_RIGHTS_WRITE	Standard Access Rights
STANDARD_RIGHTS_EXECUTE	Standard Access Rights
STANDARD_RIGHTS_ALL	Standard Access Rights
SPECIFIC_RIGHTS_ALL	ACCESS_MASK
ACCESS_SYSTEM_SECURITY	ACCESS_MASK
MAXIMUM_ALLOWED	ACCESS_MASK
GENERIC_READ	ACCESS_MASK
GENERIC_WRITE	ACCESS_MASK
GENERIC_EXECUTE	ACCESS_MASK
GENERIC_ALL	ACCESS_MASK

dwShareMode

[in] 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.

Value	Meaning
FILE_SHARE_DELETE	**Windows NT/2000:** 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

[in] 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/2000: 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.

dwCreationDisposition

[in] 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.

Value	Meaning
CREATE_NEW	Creates a new file. The function fails if the specified file already exists.
CREATE_ALWAYS	Creates a new file. If the file exists, the function overwrites the file and clears the existing attributes.
OPEN_EXISTING	Opens the file. The function fails if the file does not exist. For a discussion of why you should use the OPEN_EXISTING flag if you are using the **CreateFile** function for devices, see Remarks.
OPEN_ALWAYS	Opens the file, if it exists. If the file does not exist, the function creates the file as if *dwCreationDisposition* 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

[in] 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.

Attribute	Meaning

FILE_ATTRIBUTE_ARCHIVE	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. This flag has no effect if FILE_ATTRIBUTE_SYSTEM is also specified.
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_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
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.

Flag	Meaning
FILE_FLAG_WRITE_THROUGH	Instructs the system to write through any intermediate cache and go directly to disk. The system 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 file read and write 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 file read and write 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 should be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). Depending on the disk, this requirement may not be enforced. 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/2000:** Indicates that the file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks, provided it has the necessary privileges. The relevant privileges 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 16-bit Windows.
FILE_FLAG_OPEN_REPARSE_POINT	Specifying this flag inhibits the reparse behavior of NTFS reparse points. When the file is opened, a file handle is returned, whether the filter that controls the reparse point is operational or not. This flag cannot be used with the CREATE_ALWAYS flag.
FILE_FLAG_OPEN_NO_RECALL	Indicates that the file data is requested, but it should continue to reside in remote storage. It should not be transported back to local storage. This flag is intended for use by remote storage systems or the Hierarchical Storage Management system.

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

Value	Meaning

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.

hTemplateFile

[in] 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: The *hTemplateFile* parameter must be NULL. If you supply a handle, 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 *dwCreationDisposition* 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.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Files

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.

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 *dwCreationDisposition*.

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.

Windows NT/2000: If you rename or delete a file, then restore it shortly thereafter, Windows NT searches the cache for file information to restore. Cached information includes its short/long name pair and creation time.

Windows NT/2000: Some file systems, such as NTFS, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new file inherits the compression and encryption attributes of its directory.

You cannot use the **CreateFile** function to set a file's compression state. Use the **DeviceIoControl** function to set a file's compression state.

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 *dwCreationDisposition* 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

Volume handles may be opened as noncached at the discretion of the file system, even when the noncached option is not specified with **CreateFile**. You should assume that all Microsoft file systems open volume handles as noncached. The restrictions on noncached I/O for files apply to volumes as well.

A file system may or may not require buffer alignment even though the data is noncached. However, if the noncached option is specified when opening a volume, buffer alignment is enforced regardless of the file system on the volume. It is recommended on all file systems that you open volume handles as noncached and follow the noncached I/O restrictions.

Windows NT/2000: 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 **\\.\PHYSICALDRIVE*x*** to open the hard disk *x*. Hard disk numbers start at zero. For example:

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

For an example showing how to open a physical drive, see Calling DeviceIoControl on Windows NT/2000.

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

String	Meaning
\\.**A:**	Obtains a handle to drive A on the user's computer.
\\.**C:**	Obtains a handle to drive C on the user's computer.

There is no trailing backslash in a drive name. The string "\\.\c:\" refers to the root directory of drive C.

On Windows 2000, you can also open a volume by referring to its unique volume name. In this case also, there should be no trailing backslash on the unique volume name.

Note that all I/O buffers should be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size), even if the disk device is opened without the FILE_FLAG_NO_BUFFERING flag. Depending the disk, this requirement may not be enforced.

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 *dwCreationDisposition* 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.

Tape Drives

Windows NT/2000: You can open tape drives using a file name of the form **\\.\TAPE***x* where *x* is a number indicating which drive to open, starting with tape drive 0. To open tape drive 0 in C, use the file name "\\\\.\\TAPE0". For more information on manipulating tape drives for backup or other applications, see Tape Backup.

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.

Parameters	Value
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.
dwCreationDisposition	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/2000:** 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/2000: 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 or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

You cannot use the **CreateFile** function to set a directory's compression state. Use the **DeviceIoControl** function to set a directory's compression state.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, ACCESS_MASK, AllocConsole, CloseHandle, ConnectNamedPipe, CreateDirectory, CreateDirectoryEx, CreateNamedPipe, DeviceIOControl, GetDiskFreeSpace, GetOverlappedResult, GetStdHandle, OpenFile, OVERLAPPED, ReadFile, SECURITY_ATTRIBUTES, SetErrorMode, SetStdHandle, Standard Access Rights, TransactNamedPipe, Unique Volume Names, VirtualAlloc, WriteFile

1.30 CreateFileMapping

The **CreateFileMapping** function creates or opens a named or unnamed file-mapping object for the specified file.

```
CreateFileMapping: procedure
(
    hFile:                      dword;
    lpFileMappingAttributes:    dword;  // Should be NULL
    flProtect:                  dword;
    dwMaximumSizeHigh:          dword;
    dwMaximumSizeLow:           dword;
    lpName:                     string
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateFileMappingA@24" );
```

Parameters

hFile

[in] Handle to the file from which to create a mapping object. The file must be opened with an access mode compatible with the protection flags specified by the *flProtect* parameter. It is recommended, though not required, that files you intend to map be opened for exclusive access.

If *hFile* is INVALID_HANDLE_VALUE, the calling process must also specify a mapping object size in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this case, **CreateFileMapping** creates a file-mapping object of the specified size backed by the operating-system paging file rather than by a named file in the file system. The file-mapping object can be shared through duplication, through inheritance, or by name.

lpAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new file-mapping object. If *lpAttributes* is NULL, the file-mapping object gets a default security descriptor.

flProtect

[in] Protection desired for the file view, when the file is mapped. This parameter can be one of the following values.

Value	Description
PAGE_READONLY	Gives read-only access to the committed region of pages. An attempt to write to or execute the committed region results in an access violation. The file specified by the *hFile* parameter must have been created with GENERIC_READ access.
PAGE_READWRITE	Gives read/write access to the committed region of pages. The file specified by *hFile* must have been created with GENERIC_READ and GENERIC_WRITE access.
PAGE_WRITECOPY	Gives copy on write access to the committed region of pages. The files specified by the *hFile* parameter must have been created with GENERIC_READ and GENERIC_WRITE access.

In addition, an application can specify certain section attributes by combining (using the bitwise OR operator) one or more of the following section attribute values with one of the preceding page protection values.

Value	Description
SEC_COMMIT	Allocates physical storage in memory or in the paging file on disk for all pages of a section. This is the default setting.
SEC_IMAGE	The file specified for a section's file mapping is an executable image file. Because the mapping information and file protection are taken from the image file, no other attributes are valid with SEC_IMAGE.
SEC_NOCACHE	All pages of a section are to be set as noncacheable. This attribute is intended for architectures requiring various locking structures to be in memory that is never fetched into the processor's. On 80x86 and MIPS machines, using the cache for these structures only slows down the performance as the hardware keeps the caches coherent. Some device drivers require noncached data so that programs can write through to the physical memory. SEC_NOCACHE requires either the SEC_RESERVE or SEC_COMMIT to also be set.
SEC_RESERVE	Reserves all pages of a section without allocating physical storage. The reserved range of pages cannot be used by any other allocation operations until it is released. Reserved pages can be committed in subsequent calls to the **VirtualAlloc** function. This attribute is valid only if the *hFile* parameter is INVALID_HANDLE_VALUE; that is, a file-mapping object backed by the operating system paging file.

dwMaximumSizeHigh

[in] High-order **DWORD** of the maximum size of the file-mapping object.

dwMaximumSizeLow

[in] Low-order **DWORD** of the maximum size of the file-mapping object. If this parameter and *dwMaximumSizeHigh* are zero, the maximum size of the file-mapping object is equal to the current size of the file identified by *hFile*.

An attempt to map a file with a length of zero in this manner fails with an error code of ERROR_FILE_INVALID. Applications should test for files with a length of zero and reject such files.

lpName

[in] Pointer to a null-terminated string specifying the name of the mapping object.

If this parameter matches the name of an existing named mapping object, the function requests access to the mapping object with the protection specified by *flProtect*.

If this parameter is NULL, the mapping object is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the file-mapping object. If the object existed before the function call, the function returns a handle to the existing object (with its current size, not the specified size) and **GetLas-**

tError returns ERROR_ALREADY_EXISTS.

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

Remarks

After a file-mapping object has been created, the size of the file must not exceed the size of the file-mapping object; if it does, not all of the file's contents will be available for sharing.

If an application specifies a size for the file-mapping object that is larger than the size of the actual named file on disk, the file on disk is grown to match the specified size of the file-mapping object. If the file cannot be grown, this results in a failure to create the file-mapping object. **GetLastError** will return ERROR_DISK_FULL.

The handle that **CreateFileMapping** returns has full access to the new file-mapping object. It can be used with any function that requires a handle to a file-mapping object. File-mapping objects can be shared either through process creation, through handle duplication, or by name. For information on duplicating handles, see **DuplicateHandle**. For information on opening a file-mapping object by name, see **OpenFileMapping**.

Windows 95: File handles that have been used to create file-mapping objects must *not* be used in subsequent calls to file I/O functions, such as **ReadFile** and **WriteFile**. In general, if a file handle has been used in a successful call to the **CreateFileMapping** function, do not use that handle unless you first close the corresponding file-mapping object.

Creating a file-mapping object creates the potential for mapping a view of the file but does not map the view. The **MapViewOfFile** and **MapViewOfFileEx** functions map a view of a file into a process's address space.

With one important exception, file views derived from a single file-mapping object are coherent, or identical, at a given time. If multiple processes have handles of the same file-mapping object, they see a coherent view of the data when they map a view of the file.

The exception has to do with remote files. Although **CreateFileMapping** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer will only see its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped file and a file accessed by means of the input and output (I/O) functions (**ReadFile** and **WriteFile**) are not necessarily coherent.

To fully close a file-mapping object, an application must unmap all mapped views of the file-mapping object by calling **UnmapViewOfFile**, and close the file-mapping object handle by calling **CloseHandle**. The order in which these functions are called does not matter. The call to **UnmapViewOfFile** is necessary because mapped views of a file-mapping object maintain internal open handles to the object, and a file-mapping object will not close until all open handles to it are closed.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a memory mapped view. For more information, see Reading and Writing.

Example

To implement a mapping-object creation function that fails if the object already exists, an application can use the following code.

```
hMap = CreateFileMapping(...);

if (hMap != NULL && GetLastError() == ERROR_ALREADY_EXISTS)
{
    CloseHandle(hMap);
    hMap = NULL;
}
return hMap;
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, CloseHandle, DuplicateHandle, MapViewOfFile, MapViewOfFileEx, OpenFileMapping, ReadFile, SECURITY_ATTRIBUTES, UnmapViewOfFile, VirtualAlloc, WriteFile

---

1.31 CreateHardLink

The **CreateHardLink** function establishes an NTFS hard link between an existing file and a new file. An NTFS hard link is similar to a POSIX hard link.

```
CreateHardLink: procedure
(
        lpFileName:             string;
        lpExistingFileName:     string;
    var lpSecurityAttributes:   Security_Attributes
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateHardLinkA@12" );
```

Parameters

lpFileName

[in] Pointer to the name of the new directory entry to be created.

lpExistingFileName

[in] Pointer to the name of the existing file to which the new link will point.

lpSecurityAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new file.

If this parameter is NULL, it leaves the file's existing security descriptor unmodified.

If this parameter is not NULL, it modifies the file's security descriptor.

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

Any directory entry for a file, whether created with **CreateFile** or **CreateHardLink**, is a hard link to the associated file. Additional hard links, created with the **CreateHardLink** function, allow you to have multiple directory entries for a file, that is, multiple hard links to the same file. These may be different names in the same directory, or they may be the same (or different) names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are just directory entries for a file, whenever an application modifies a file through any hard link, all applications using any other hard link to the file see the changes. Also, all of the directory entries are updated if the file changes. For example, if the file's size changes, all of the hard links to the file will show the new size.

The security descriptor belongs to the file to which the hard link points. The link itself, being merely a directory entry, has no security descriptor. Thus, if you change the security descriptor of any hard link, you're actually changing the underlying file's security descriptor. All hard links that point to the file will thus allow the newly specified access. There is no way to give a file different security descriptors on a per-hard-link basis.

Use **DeleteFile** to delete hard links. You can delete them in any order regardless of the order in which they were created.

Flags, attributes, access, and sharing as specified in **CreateFile** operate on a per-file basis. That is, if you open a file with no sharing allowed, another application cannot share the file by creating a new hard link to the file.

CreateHardLink does not work over the network redirector.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows 2000.

See Also

File Systems Overview, File System Functions, CreateFile, DeleteFile, SECURITY_ATTRIBUTES

1.32 CreateIoCompletionPort

The **CreateIoCompletionPort** function can associate an instance of an opened file with a newly created or an existing input/output (I/O) completion port, or it can create an I/O completion port without associating it with a file.

Associating an instance of an opened file with an I/O completion port lets an application receive notification of the completion of asynchronous I/O operations involving that file.

```
CreateIoCompletionPort: procedure
(
        FileHandle:                 dword;
        ExistingCompletionPort:     dword;
    var CompletionKey:              dword;
        NumberOfConcurrentThreads:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateIoCompletionPort@16" );
```

Parameters

FileHandle

[in] Handle to a file opened for overlapped I/O completion. You must specify the FILE_FLAG_OVERLAPPED flag when using the **CreateFile** function to obtain the handle.

If *FileHandle* specifies INVALID_HANDLE_VALUE, **CreateIoCompletionPort** creates an I/O completion port without associating it with a file. In this case, the *ExistingCompletionPort* parameter must be NULL and the *CompletionKey* parameter is ignored.

ExistingCompletionPort

[in] Handle to the I/O completion port.

If this parameter specifies an existing completion port, the function associates it with the file specified by the *FileHandle* parameter. The function returns the handle of the existing completion port; it does not create a new I/O completion port.

If this parameter is NULL, the function creates a new I/O completion port and associates it with the file specified by *FileHandle*. The function returns the handle to the new I/O completion port.

CompletionKey

[in] Per-file completion key that is included in every I/O completion packet for the specified file.

NumberOfConcurrentThreads

[in] Maximum number of threads that the operating system allows to concurrently process I/O completion packets for the I/O completion port. If this parameter is zero, the system allows as many concurrently running threads as there are processors in the system.

Although any number of threads can call the **GetQueuedCompletionStatus** function to wait for an I/O completion port, each thread is associated with only one completion port at a time. That port is the port that was last checked by the thread.

When a packet is queued to a port, the system first checks how many threads associated with the port are running. If the number of threads running is less than the value of *NumberOfConcurrentThreads*, then one of the waiting threads is allowed to process the packet. When a running thread completes its processing, it calls **GetQueuedCompletionStatus** again, at which point the system can allow another waiting thread to process a packet.

The system also allows a waiting thread to process a packet if a running thread enters any wait state. When the thread in the wait state begins running again, there may be a brief period when the number of active threads exceeds the *NumberOfConcurrentThreads* value. However, the system quickly reduces the number by not allowing any new active threads until the number of active threads falls below the specified value.

Return Values

If the function succeeds, the return value is the handle to the I/O completion port that is associated with the specified file.

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

Remarks

The I/O system can be instructed to send I/O completion notification packets to I/O completion ports, where they are queued. The **CreateIoCompletionPort** function provides this functionality.

After an instance of an open file is associated with an I/O completion port, it cannot be used in the **ReadFileEx** or **WriteFileEx** function. It is best not to share such an associated file through either handle inheritance or a call to the **DuplicateHandle** function. Operations performed with such duplicate handles generate completion notifications.

When you perform an I/O operation with a file handle that has an associated I/O completion port, the I/O system sends a completion notification packet to the completion port when the I/O operation completes. The I/O completion port places the completion packet in a first-in-first-out queue. Use the GetQueuedCompletionStatus function to retrieve these queued I/O completion packets.

Threads in the same process can use the PostQueuedCompletionStatus function to place I/O completion notification packets in a completion port's queue. By doing so, you can use the port to receive communications from other threads of the process, in addition to receiving I/O completion notification packets from the I/O system.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, DuplicateHandle, GetQueuedCompletionStatus, PostQueuedCompletionStatus, ReadFileEx, WriteFileEx

---

1.33 CreateJobObject

The **CreateJobObject** function creates or opens a job object.

```
CreateJobObject: procedure
(
    var lpJobAttributes:    Security_Attributes;
```

```
        lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateJobObjectA@8" );
```

Parameters

lpJobAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies the security descriptor for the job object and determines whether child processes can inherit the returned handle. If *lpJobAttributes* is NULL, the job object gets a default security descriptor and the handle cannot be inherited.

lpName

[in] Pointer to a null-terminated string specifying the name of the job. The name is limited to MAX_PATH characters. Name comparison is case-sensitive.

If *lpName* is NULL, the job is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the job object. The handle has JOB_OBJECT_ALL_ACCESS access to the job object. If the object existed before the function call, the function returns a handle to the existing job object and **GetLastError** returns ERROR_ALREADY_EXISTS.

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

Remarks

When a job is created, its accounting information is initialized to zero, all limits are inactive, and there are no associated processes. To associate a process with a job, use the AssignProcessToJobObject function. To set limits for a job, use the **SetInformationJobObject** function. To query accounting information, use the **QueryInformationJobObject** function.

To close a job object handle, use the **CloseHandle** function. The job is destroyed when its last handle has been closed. If there are running processes still associated with the job when it is destroyed, they will continue to run even after the job is destroyed.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows 2000.

See Also

Processes and Threads Overview, Process and Thread Functions, AssignProcessToJobObject, CloseHandle, QueryInformationJobObject, SECURITY_ATTRIBUTES, SetInformationJobObject

1.34 CreateMailslot

The **CreateMailslot** function creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. The mailslot is local to the computer that creates it. An error occurs if a mailslot with the specified name already exists.

```
CreateMailslot: procedure
(
        lpName:                 string;
        nMaxMessageSize:        dword;
        lReadTimneout:          dword;
    var lpSecurityAttributes:   SECURITY_ATTRIBUTES
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateMailslotA@16" );
```

Parameters

lpName

[in] Pointer to a null-terminated string specifying the name of the mailslot. This name must have the following form:

\\.**mailslot**\[*path*]*name*

The *name* field must be unique. The name may include multiple levels of pseudodirectories separated by backslashes. For example, both \\.\mailslot\example_mailslot_name and \\.\mailslot\abc\def\ghi are valid names.

nMaxMessageSize

[in] Specifies the maximum size, in bytes, of a single message that can be written to the mailslots. To specify that the message can be of any size, set this value to zero.

lReadTimeout

[in] Specifies the amount of time, in milliseconds, a read operation can wait for a message to be written to the mailslot before a time-out occurs. The following values have special meanings.

Value	Meaning
0	Returns immediately if no message is present. (The system does not treat an immediate return as an error.)
MAILSLOT_WAIT_FOREVER	Waits forever for a message.

This time-out value applies to all subsequent read operations and all inherited mailslot handles.

lpSecurityAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure. The **bInheritHandle** member of the structure determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited. The **lpSecurityDescriptor** member of the structure is ignored.

Return Values

If the function succeeds, the return value is a handle to the mailslot, for use in server mailslot operations. The server side of the handle is overlapped.

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

Remarks

The mailslot exists until one of the following conditions is true:

The last (possibly inherited or duplicated) handle to it is closed using the **CloseHandle** function.

The process owning the last (possibly inherited or duplicated) handle exits.

The system uses the second method to destroy mailslots.

To write a message to a mailslot, a process uses the **CreateFile** function, specifying the mailslot name by using one of the following formats.

Format	Usage
.\mailslot****name*	Retrieves a client handle to a local mailslot.
computername\mailslot*name*	Retrieves a client handle to a remote mailslot.
domainname\mailslot*name*	Retrieves a client handle to all mailslots with the specified name in the specified domain.
*\mailslot*name*	Retrieves a client handle to all mailslots with the specified name in the system's primary domain.

If **CreateFile** specifies a domain or uses the asterisk format to specify the system's primary domain, the application cannot write more than 424 bytes at a time to the mailslot. If the application attempts to do so, the **WriteFile** function fails and **GetLastError** returns ERROR_BAD_NETPATH.

An application must specify the FILE_SHARE_READ flag when using **CreateFile** to retrieve a client handle to a mailslot.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Mailslots Overview, Mailslot Functions, CloseHandle, CreateFile, GetMailslotInfo, SECURITY_ATTRIBUTES, SetMailslotInfo, WriteFile

---

1.35 CreateMutex

The **CreateMutex** function creates or opens a named or unnamed mutex object.

```
CreateMutex: procedure
(
    var lpMutexAttributes : SECURITY_ATTRIBUTES
        bInitialOwner:      boolean;
        lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateMutexA@12" );
```

Parameters

lpMutexAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new mutex. If *lpMutexAttributes* is NULL, the mutex gets a default security descriptor.

bInitialOwner

[in] Specifies the initial owner of the mutex object. If this value is TRUE and the caller created the mutex, the calling thread obtains ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex. To determine if the caller created the mutex, see the Return Values section.

lpName

[in] Pointer to a null-terminated string specifying the name of the mutex object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named mutex object, this function requests MUTEX_ALL_ACCESS access to the existing object. In this case, the *bInitialOwner* parameter is ignored because it has already been set by the creating process. If the *lpMutexAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

If *lpName* is NULL, the mutex object is created without a name.

If *lpName* matches the name of an existing event, semaphore, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the mutex object. If the named mutex object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS. Otherwise, the caller created the mutex.

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

Remarks

The handle returned by **CreateMutex** has MUTEX_ALL_ACCESS access to the new mutex object and can be used in any function that requires a handle to a mutex object.

Any thread of the calling process can specify the mutex-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a mutex object is signaled when it is not owned by any thread. The creating thread can use the *bInitialOwner* flag to request immediate ownership of the mutex. Otherwise, a thread must use one of the wait functions to request ownership. When the mutex's state is signaled, one waiting thread is granted ownership, the mutex's state changes to nonsignaled, and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses the **ReleaseMutex** function to release its ownership.

The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execu-

tion. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **ReleaseMutex** once for each time that the mutex satisfied a wait.

Two or more processes can call **CreateMutex** to create the same named mutex. The first process actually creates the mutex, and subsequent processes open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, you should set the *bInitialOwner* flag to FALSE; otherwise, it can be difficult to be certain which process has initial ownership.

Multiple processes can have handles of the same mutex object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

A child process created by the **CreateProcess** function can inherit a handle to a mutex object if the *lpMutexAttributes* parameter of **CreateMutex** enabled inheritance.

A process can specify the mutex-object handle in a call to the **DuplicateHandle** function to create a duplicate handle that can be used by another process.

A process can specify the name of a mutex object in a call to the **OpenMutex** or **CreateMutex** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

Example

For an example that uses **CreateMutex**, see Using Mutex Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateProcess, DuplicateHandle, OpenMutex, ReleaseMutex, SECURITY_ATTRIBUTES, Object Names

---

1.36 CreateNamedPipe

The **CreateNamedPipe** function creates an instance of a named pipe and returns a handle for subsequent pipe operations. A named pipe server process uses this function either to create the first instance of a specific named pipe and establish its basic attributes or to create a new instance of an existing named pipe.

```
CreateNamedPipe: procedure
(
        lpName:                 string;
        dwOpenMode:             dword;
        dwPipeMode:             dword;
        nMaxInstances:          dword;
        nOutBufferSize:         dword;
        nInbufferSize:          dword;
        nDefaultTimeOut:        dword;
    var lpSecurityAttributes:   SECURITY_ATTRIBUTES
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateNamedPipeA@32" );
```

Parameters

lpName

[in] Pointer to the null-terminated string that uniquely identifies the pipe. The string must have the following form:

\\.\pipe*pipename*

The *pipename* part of the name can include any character other than a backslash, including numbers and special characters. The entire pipe name string can be up to 256 characters long. Pipe names are not case sensitive.

dwOpenMode

[in] Specifies the pipe access mode, the overlapped mode, the write-through mode, and the security access mode of the pipe handle.

CreateNamedPipe fails if *dwOpenMode* specifies any flags other than those listed in the following tables.

This parameter must specify one of the following pipe access mode flags. The same mode must be specified for each instance of the pipe.

Mode	Description
PIPE_ACCESS_DUPLEX	The pipe is bidirectional; both server and client processes can read from and write to the pipe. This mode gives the server the equivalent of GENERIC_READ | GENERIC_WRITE access to the pipe. The client can specify GENERIC_READ or GENERIC_WRITE, or both, when it connects to the pipe using the **`CreateFile`** function.
PIPE_ACCESS_INBOUND	The flow of data in the pipe goes from client to server only. This mode gives the server the equivalent of GENERIC_READ access to the pipe. The client must specify GENERIC_WRITE access when connecting to the pipe.
PIPE_ACCESS_OUTBOUND	The flow of data in the pipe goes from server to client only. This mode gives the server the equivalent of GENERIC_WRITE access to the pipe. The client must specify GENERIC_READ access when connecting to the pipe.

This parameter can also include either or both of the following flags, which enable write-through mode and overlapped mode. These modes can be different for different instances of the same pipe.

Mode	Description
FILE_FLAG_WRITE_THROUGH	Write-through mode is enabled. This mode affects only write operations on byte-type pipes and, then, only when the client and server processes are on different computers. If this mode is enabled, functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer. If this mode is not enabled, the system enhances the efficiency of network operations by buffering data until a minimum number of bytes accumulate or until a maximum time elapses.

FILE_FLAG_OVERLAPPED	Overlapped mode is enabled. If this mode is enabled, functions performing read, write, and connect operations that may take a significant time to be completed can return immediately. This mode enables the thread that started the operation to perform other operations while the time-consuming operation executes in the background. For example, in overlapped mode, a thread can handle simultaneous input and output (I/O) operations on multiple instances of a pipe or perform simultaneous read and write operations on the same pipe handle. If overlapped mode is not enabled, functions performing read, write, and connect operations on the pipe handle do not return until the operation is finished. The **ReadFileEx** and **WriteFileEx** functions can only be used with a pipe handle in overlapped mode. The **ReadFile**, **WriteFile, ConnectNamedPipe,** and **TransactNamedPipe** functions can execute either synchronously or as overlapped operations.

This parameter can include any combination of the following security access mode flags. These modes can be different for different instances of the same pipe. They can be specified without concern for what other *dwOpenMode* modes have been specified.

Mode	Description
WRITE_DAC	The caller will have write access to the named pipe's discretionary access control list (ACL).
WRITE_OWNER	The caller will have write access to the named pipe's owner.
ACCESS_SYSTEM_SECURITY	The caller will have write access to the named pipe’s SACL. For more information, see Access-Control Lists (ACLs) and SACL Access Right.

dwPipeMode

[in] Specifies the type, read, and wait modes of the pipe handle.

One of the following type mode flags can be specified. The same type mode must be specified for each instance of the pipe. If you specify zero, the parameter defaults to byte-type mode.

Mode	Description
PIPE_TYPE_BYTE	Data is written to the pipe as a stream of bytes. This mode cannot be used with PIPE_READMODE_MESSAGE.
PIPE_TYPE_MESSAGE	Data is written to the pipe as a stream of messages. This mode can be used with either PIPE_READMODE_MESSAGE or PIPE_READMODE_BYTE.

One of the following read mode flags can be specified. Different instances of the same pipe can specify different read modes. If you specify zero, the parameter defaults to byte-read mode.

Mode	Description
PIPE_READMODE_BYTE	Data is read from the pipe as a stream of bytes. This mode can be used with either PIPE_TYPE_MESSAGE or PIPE_TYPE_BYTE.
PIPE_READMODE_MESSAGE	Data is read from the pipe as a stream of messages. This mode can be only used if PIPE_TYPE_MESSAGE is also specified.

One of the following wait mode flags can be specified. Different instances of the same pipe can specify different

wait modes. If you specify zero, the parameter defaults to blocking mode.

Mode	Description
PIPE_WAIT	Blocking mode is enabled. When the pipe handle is specified in the **ReadFile, WriteFile,** or **ConnectNamedPipe** function, the operations are not completed until there is data to read, all data is written, or a client is connected. Use of this mode can mean waiting indefinitely in some situations for a client process to perform an action.
PIPE_NOWAIT	Nonblocking mode is enabled. In this mode, **ReadFile**, **WriteFile**, and **ConnectNamedPipe** always return immediately. Note that nonblocking mode is supported for compatibility with Microsoft LAN Manager version 2.0 and should not be used to achieve asynchronous I/O with named pipes. For more information on asynchronous pipe I/O, see Synchronous and Overlapped Input and Output.

nMaxInstances

[in] Specifies the maximum number of instances that can be created for this pipe. The same number must be specified for all instances. Acceptable values are in the range 1 through PIPE_UNLIMITED_INSTANCES. If this parameter is PIPE_UNLIMITED_INSTANCES, the number of pipe instances that can be created is limited only by the availability of system resources.

nOutBufferSize

[in] Specifies the number of bytes to reserve for the output buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

nInBufferSize

[in] Specifies the number of bytes to reserve for the input buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

nDefaultTimeOut

[in] Specifies the default time-out value, in milliseconds, if the **WaitNamedPipe** function specifies NMPWAIT_USE_DEFAULT_WAIT. Each instance of a named pipe must specify the same value.

lpSecurityAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new named pipe and determines whether child processes can inherit the returned handle. If *lpSecurityAttributes* is NULL, the named pipe gets a default security descriptor and the handle cannot be inherited.

Return Values

If the function succeeds, the return value is a handle to the server end of a named pipe instance.

If the function fails, the return value is INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**. The return value is ERROR_INVALID_PARAMETER if *nMaxInstances* is greater than PIPE_UNLIMITED_INSTANCES.

Remarks

To create an instance of a named pipe by using **CreateNamedPipe**, the user must have FILE_CREATE_PIPE_INSTANCE access to the named pipe object. If a new named pipe is being created, the access control list (ACL) from the security attributes parameter defines the discretionary access control for the named pipe.

All instances of a named pipe must specify the same pipe type (byte-type or message-type), pipe access (duplex, inbound, or outbound), instance count, and time-out value. If different values are used, this function fails and **GetLastError** returns ERROR_ACCESS_DENIED.

The client side of a named pipe starts out in byte mode, even if the server side is in message mode. To avoid problems receiving data, set the client side to message mode as well.

The input and output buffer sizes are advisory. The actual buffer size reserved for each end of the named pipe is either the system default, the system minimum or maximum, or the specified size rounded up to the next allocation boundary.

The pipe server should not perform a blocking read operation until the pipe client has started. Otherwise, a race condition can occur. This typically occurs when initialization code, such as the C run-time, needs to lock and examine inherited handles.

An instance of a named pipe is always deleted when the last handle to the instance of the named pipe is closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, ConnectNamedPipe, CreateFile, ReadFile, ReadFileEx, SECURITY_ATTRIBUTES, TransactNamedPipe, WaitNamedPipe, WriteFile, WriteFileEx

---

1.37 CreatePipe

The **CreatePipe** function creates an anonymous pipe, and returns handles to the read and write ends of the pipe.

```
CreatePipe: procedure
(
    var hReadPipe:          dword;
    var hWritePipe:         dword;
    var lpPipeAttributes:   SECURITY_ATTRIBUTES;
    nSize:                  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreatePipe@16" );
```

Parameters

hReadPipe

[out] Pointer to a variable that receives the read handle for the pipe.

hWritePipe

[out] Pointer to a variable that receives the write handle for the pipe.

lpPipeAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new pipe. If *lpPipeAttributes* is NULL, the pipe gets a default security descriptor.

nSize

[in] Specifies the buffer size for the pipe, in bytes. The size is only a suggestion; the system uses the value to calculate an appropriate buffering mechanism. If this parameter is zero, the system uses the default buffer size.

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

CreatePipe creates the pipe, assigning the specified pipe size to the storage buffer. **CreatePipe** also creates handles that the process uses to read from and write to the buffer in subsequent calls to the **ReadFile** and **WriteFile** functions.

To read from the pipe, a process uses the read handle in a call to the **ReadFile** function. **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.

When a process uses **WriteFile** to write to an anonymous pipe, the write operation is not completed until all bytes are written. If the pipe buffer is full before all bytes are written, **WriteFile** does not return until another process or thread uses **ReadFile** to make more buffer space available.

Windows NT/2000: Anonymous pipes are implemented using a named pipe with a unique name. Therefore, you can often pass a handle to an anonymous pipe to a function that requires a handle to a named pipe.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, ReadFile, SECURITY_ATTRIBUTES, WriteFile

---

1.38 CreateProcess

The **CreateProcess** function creates a new process and its primary thread. The new process runs the specified executable file.

To create a process that runs in a different security context, use the **CreateProcessAsUser** or **CreateProcessWithLogonW** function.

```
CreateProcess: procedure
(
        lpApplicationName:      string;
        lpCommandLine:          string;
    var lpProcessAttributes:    SECURITY_ATTRIBUTES;
    var lpThreadAttributes:     SECURITY_ATTRIBUTES;
        InheritHandles:         boolean;
        dwCreationFlags:        dword;
    var lpEnvironment:          var;
        lpCurrentyDirectory:    string;
    var lpStartupInfo:          STARTUPINFO;
    var lpProcessInformation:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateProcessA@40" );
```

Parameters

lpApplicationName

[in] Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe

The specified module can be a Win32-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

Windows NT/2000: If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments. A 16-bit application is one that executes as a VDM or WOW process.

lpCommandLine

[in] Pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line, trimming the string if necessary, to indicate which file was actually used.

Windows NT/2000: The Unicode version of this function, **CreateProcessW**, will fail if this parameter is a const string.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use **`GetCommandLine`** to retrieve the entire command line. C runtime processes can use the **argc** and **argv** arguments. Note that it is a common practice to repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white-space – delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

The directory from which the application loaded.

The current directory for the parent process.

Windows 95/98: The Windows system directory. Use the **`GetSystemDirectory`** function to get the path of this directory.

Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is System32.

Windows NT/2000: 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.

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

The directories that are listed in the PATH environment variable.

lpProcessAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new process. If *lpProcessAttributes* is NULL, the process gets a default security descriptor.

lpThreadAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the main thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

bInheritHandles

[in] Indicates whether the new process inherits handles from the calling process. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

dwCreationFlags

[in] Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted.

Value	Meaning
CREATE_BREAKAWAY_FROM_JOB	**Windows 2000:** The child processes of a process associated with a job are not associated with the job. If the calling process is not associated with a job, this flag has no effect. If the calling process is associated with a job, the job must set the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit or **CreateProcess** will fail.
CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, **CreateProcess** gives the new process the current default error mode. An application sets the current default error mode by calling **SetErrorMode**. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for **CreateProcess** is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_FORCEDOS	**Windows NT/2000:** This flag is valid only when starting a 16-bit bound application. If set, the system will force the application to run as an MS-DOS-based application rather than as an OS/2-based application.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag.

CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the *lpProcessInformation* parameter. Process groups are used by the **`GenerateConsoleCtrlEvent`** function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes.
CREATE_NO_WINDOW	**Windows NT/2000:** This flag is valid only when starting a console application. If set, the console application is run without a console window.
CREATE_SEPARATE_WOW_VDM	**Windows NT/2000:** This flag is valid only when starting a 16-bit Windows-based application. If set, the new process runs in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications run as threads in a single, shared VDM. The advantage of running separately is that a crash only terminates the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that are run in separate VDMs have separate input queues. That means that if one application stops responding momentarily, applications in separate VDMs continue to receive input. The disadvantage of running separately is that it takes significantly more memory to do so. You should use this flag only if the user requests that 16-bit applications should run in them own VDM.
CREATE_SHARED_WOW_VDM	**Windows NT/2000:** The flag is valid only when starting a 16-bit Windows-based application. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is TRUE, this flag causes the **CreateProcess** function to override the switch and run the new process in the shared Virtual DOS Machine.
CREATE_SUSPENDED	The primary thread of the new process is created in a suspended state, and does not run until the **`ResumeThread`** function is called.
CREATE_UNICODE_ENVIRONMENT	Indicates the format of the *lpEnvironment* parameter. If this flag is set, the environment block pointed to by *lpEnvironment* uses Unicode characters. Otherwise, the environment block uses ANSI characters.
DEBUG_PROCESS	If this flag is set, the calling process is treated as a debugger, and the new process is debugged. The system notifies the debugger of all debug events that occur in the process being debugged. If you create a process with this flag set, only the calling thread (the thread that called **CreateProcess**) can call the **`WaitForDebugEvent`** function. **Windows 95/98:** This flag is not valid if the new process is a 16-bit application.
DEBUG_ONLY_THIS_PROCESS	If this flag is not set and the calling process is being debugged, the new process becomes another process being debugged by the calling process's debugger. If the calling process is not a process being debugged, no debugging-related actions occur.

DETACHED_PROCESS	For console processes, the new process does not have access to the console of the parent process. The new process can call the **AllocConsole** function at a later time to create a new console. This flag cannot be used with the CREATE_NEW_CONSOLE flag.

The *dwCreationFlags* parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. If none of the following priority class flags is specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process. You can specify one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than NORMAL_PRIORITY_CLASS but lower than HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than IDLE_PRIORITY_CLASS but lower than NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the system. Use extreme care when using the high-priority class, because a CPU-bound application with a high-priority class can use nearly all available cycles.
IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

lpEnvironment

[in] Pointer to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

```
name=value
```

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, set the *dwCreationFlags* field's CREATE_UNICODE_ENVIRONMENT flag. Otherwise, do not set this flag.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

lpCurrentDirectory

[in] Pointer to a null-terminated string that specifies the current drive and directory for the child process. The string must be a full path and file name that includes a drive letter. If this parameter is NULL, the new process will have the same current drive and directory as the calling process. This option is provided primarily for shells that need to start an application and specify its initial drive and working directory.

lpStartupInfo

[in] Pointer to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lpProcessInformation

[out] Pointer to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

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 **CreateProcess** function is used to run a new program. The **WinExec** and **LoadModule** functions are still available, but they are implemented as calls to **CreateProcess**.

In addition to creating a process, **CreateProcess** also creates a thread object. The thread is created with an initial stack whose size is described in the image header of the specified program's executable file. The thread begins execution at the image's entry point.

When created, the new process and the new thread handles receive full access rights. For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle to that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the thread.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the **OpenProcess** function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the **PROCESS_INFORMATION** structure.

When specifying an application name in the *lpApplicationName* or *lpCommandLine* strings, it doesn't matter whether the application name includes the file name extension, with one exception: an MS-DOS – based or Windows-based application whose file name extension is .com must include the .com extension.

The calling thread can use the **WaitForInputIdle** function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcess** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the **ExitProcess** function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs).

ExitProcess, ExitThread, CreateThread, CreateRemoteThread, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events at a time can happen in an address space, and the following restrictions apply.

During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.

Only one thread at a time can be in a DLL initialization or detach routine.

The **ExitProcess** function does not return until there are no threads are in their DLL initialization or detach routines.

The created process remains in the system until all threads within the process have terminated and all handles to the process and any of its threads have been closed through calls to **CloseHandle**. The handles for both the process and the main thread must be closed through calls to **CloseHandle**. If these handles are not needed, it is best to close them immediately after the process is created.

When the last thread in a process terminates, the following events occur:

All objects opened by the process are implicitly closed.

The process's termination status (which is returned by **GetExitCodeProcess**) changes from its initial value of STILL_ACTIVE to the termination status of the last thread to terminate.

The thread object of the main thread is set to the signaled state, satisfying any threads that were waiting on the object.

The process object is set to the signaled state, satisfying any threads that were waiting on the object.

If the current directory on drive C is \MSVC\MFC, there is an environment variable called =C: whose value is C:\MSVC\MFC. As noted in the previous description of *lpEnvironment*, such current directory information for a system's drives does not automatically propagate to a new process when the **CreateProcess** function's *lpEnvironment* parameter is non-NULL. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create the =X environment variable strings, get them into alphabetical order (because the system uses a sorted environment), and then put them into the environment block specified by *lpEnvironment*. Typically, they will go at the front of the environment block, due to the previously mentioned environment block sorting.

One way to obtain the current directory variable for a drive X is to call **GetFullPathName**("X:",. .). That avoids an application having to scan the environment block. If the full path returned is X:\, there is no need to pass that value on as environment data, since the root directory is the default current directory for drive X of a new process.

The handle returned by the **CreateProcess** function has PROCESS_ALL_ACCESS access to the process object.

The current directory specified by the *lpcurrentDirectory* parameter is the current directory for the child process. The current directory specified in item 2 under the *lpCommandLine* parameter is the current directory for the parent process.

Note The name of the executable in the command line that the operating system provides to a process is not necessarily identical to that in the command line that the calling process gives to the **CreateProcess** function. The operating system may prepend a fully qualified path to an executable name that is provided without a fully qualified path.

Windows NT/2000: When a process is created with CREATE_NEW_PROCESS_GROUP specified, an implicit call to **SetConsoleCtrlHandler**(NULL,TRUE) is made on behalf of the new process; this means that the new process has CTRL+C disabled. This lets good shells handle CTRL+C themselves, and selectively pass that signal on to sub-processes. CTRL+BREAK is not disabled, and may be used to interrupt the process/process group.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Processes and Threads Overview, Process and Thread Functions, AllocConsole, CloseHandle, CreateProcessAsUser, CreateProcessWithLogonW, CreateRemoteThread, CreateThread, ExitProcess, ExitThread, GenerateConsoleCtrlEvent, GetCommandLine, GetEnvironmentStrings, GetExitCodeProcess, GetFullPathName, GetStartupInfo, GetSystemDirectory, GetWindowsDirectory, LoadModule, OpenProcess, PROCESS_INFORMATION, ResumeThread, SECURITY_ATTRIBUTES, SetConsoleCtrlHandler, SetErrorMode, STARTUPINFO, TerminateProcess, WaitForInputIdle, WaitForDebugEvent, WinExec

1.39 CreateRemoteThread

The **CreateRemoteThread** function creates a thread that runs in the virtual address space of another process.

```
CreateRemoteThread: procedure
(
        hProcess:           dword;
    var lpThreadAttributes: SECURITY_ATTRIBUTES;
        dwStackSize:        dword;
        lpStartAddress:     LPTHREAD_START_ROUTINE;
        lpParameter:        dword;
        dwCreationFlags:    dword;
    var lpThreadId:         dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateRemoteThread@28" );
```

Parameters

hProcess

[in] Handle to the process in which the thread is to be created. The handle must have the PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, and PROCESS_VM_READ access rights. For more information, see Process Security and Access Rights.

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited.

dwStackSize

[in] Specifies the initial commit size of the stack, in bytes. The system rounds this value to the nearest page. If this value is zero, or is smaller than the default commit size, the default is to use the same size as the calling thread. For more information, see **Thread Stack Size.**

lpStartAddress

[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process. For more information on the thread function, see **ThreadProc.**

lpParameter

[in] Specifies a single value passed to the thread function.

dwCreationFlags

[in] Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation.

lpThreadId

[out] Pointer to a variable that receives the thread identifier.

If this parameter is NULL, the thread identifier is not returned.

Return Values

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

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

Note that **CreateRemoteThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Remarks

The **CreateRemoteThread** function causes a new thread of execution to begin in the address space of the specified process. The thread has access to all objects opened by the process.

The new thread handle is created with full access to the new thread. If a security descriptor is not provided, the handle may be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **GetThreadPriority** and **SetThreadPriority** functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting for the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a **CreateProcess** call) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.

Only one thread in a process can be in a DLL initialization or detach routine at a time.

ExitProcess does not return until no threads are in their DLL initialization or detach routines.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CloseHandle, CreateProcess, CreateThread, ExitProcess, ExitThread, GetThreadPriority, ResumeThread, SECURITY_ATTRIBUTES, SetThreadPriority, ThreadProc

---

1.40 CreateSemaphore

The **CreateSemaphore** function creates or opens a named or unnamed semaphore object.

```
CreateSemaphore: procedure
(
    var lpSemaphoreAttributes:  SECURITY_ATTRIBUTES;
        lInitialCount:          int32;
        lMaximumCount:          int32;
        lpName:                 lpName
```

```
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateSemaphoreA@16" );
```

Parameters

lpSemaphoreAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new semaphore. If *lpSemaphoreAttributes* is NULL, the semaphore gets a default security descriptor.

lInitialCount

[in] Specifies an initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to *lMaximumCount*. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the **ReleaseSemaphore** function.

lMaximumCount

[in] Specifies the maximum count for the semaphore object. This value must be greater than zero.

lpName

[in] Pointer to a null-terminated string specifying the name of the semaphore object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named semaphore object, this function requests SEMAPHORE_ALL_ACCESS access to the existing object. In this case, the *lInitialCount* and *lMaximumCount* parameters are ignored because they have already been set by the creating process. If the *lpSemaphoreAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

If *lpName* is NULL, the semaphore object is created without a name.

If *lpName* matches the name of an existing event, mutex, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS.

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

Remarks

The handle returned by **CreateSemaphore** has SEMAPHORE_ALL_ACCESS access to the new semaphore object and can be used in any function that requires a handle to a semaphore object.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a semaphore object is signaled when its count is greater than zero, and nonsignaled when its count is equal to zero. The *lInitialCount* parameter specifies the initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one. Use the **`ReleaseSemaphore`** function to increment a semaphore's count by a specified amount. The count can never be less than zero or greater than the value specified in the *lMaximumCount* parameter.

Multiple processes can have handles of the same semaphore object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

A child process created by the **`CreateProcess`** function can inherit a handle to a semaphore object if the *lpSemaphoreAttributes* parameter of **CreateSemaphore** enabled inheritance.

A process can specify the semaphore-object handle in a call to the **`DuplicateHandle`** function to create a duplicate handle that can be used by another process.

A process can specify the name of a semaphore object in a call to the **OpenSemaphore** or **CreateSemaphore** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

Example

For an example that uses **CreateSemaphore**, see Using Semaphore Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.lib.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateProcess, DuplicateHandle, OpenSemaphore, ReleaseSemaphore, SECURITY_ATTRIBUTES, Object Names

---

1.41 CreateTapePartition

The **CreateTapePartition** function reformats a tape.

```
CreateTapePartition: procedure
(
    hDevice:            dword;
    dwParitionMethod:   dword;
    dwCount:            dword;
    dwSize:             dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateTapePartition@16" );
```

Parameters

hDevice

[in] Handle to the device where the new partition is to be created. This handle is created by using the **CreateFile** function.

dwPartitionMethod

[in] Specifies the type of partition to create. To determine what type of partitions your device supports, see the documentation for your hardware. This parameter can have one of the following values.

Value	Description
TAPE_FIXED_PARTITIONS	Partitions the tape based on the device's default definition of partitions. The *dwCount* and *dwSize* parameters are ignored.
TAPE_INITIATOR_PARTITIONS	Partitions the tape into the number and size of partitions specified by *dwCount* and *dwSize*, respectively, except for the last partition. The size of the last partition is the remainder of the tape.
TAPE_SELECT_PARTITIONS	Partitions the tape into the number of partitions specified by *dwCount*. The *dwSize* parameter is ignored. The size of the partitions is determined by the device's default partition size. For more specific information, refer to the documentation for your tape device.

dwCount

[in] Specifies the number of partitions to create. The **GetTapeParameters** function provides the maximum number of partitions a tape can support.

dwSize

[in] Specifies the size, in megabytes, of each partition. This value is ignored if the *dwPartitionMethod* parameter is TAPE_SELECT_PARTITIONS.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, it may return one of the following error codes:

Error	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.

ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

Creating partitions reformats the tape. All previous information recorded on the tape is destroyed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, GetTapeParameters

---

1.42 CreateThread

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.

To create a thread that runs in the virtual address space of another process, use the **CreateRemoteThread** function.

```
CreateThread: procedure
(
    var lpThreadAttributes: SECURITY_ATTRIBUTES;
        dwStackSize:        dword;
        lpStartAddress:     LPTHREAD_START_ROUTINE;
        lpParameter:        dword;
        dwCreationFlags:    dword;
    var lpThreadID:         dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateThread@24" );
```

Parameters

lpThreadAttributes

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

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

dwStackSize

[in] Specifies the initial commit size of the stack, in bytes. The system rounds this value to the nearest page. If this value is zero, or is smaller than the default commit size, the default is to use the same size as the calling thread. For more information, see Thread Stack Size.

lpStartAddress

[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread. For more information on the thread function, see **`ThreadProc`**.

lpParameter

[in] Specifies a single parameter value passed to the thread.

dwCreationFlags

[in] Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the **`ResumeThread`** function is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

lpThreadId

[out] Pointer to a variable that receives the thread identifier.

Windows NT/2000: If this parameter is NULL, the thread identifier is not returned.

Windows 95/98: This parameter may not be NULL.

Return Values

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

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

Note that **CreateThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Windows 95/98: CreateThread succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

Remarks

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2028 threads. If you reduce the default stack size, you can create more threads. However, your application will have better performance if you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

The new thread handle is created with THREAD_ALL_ACCESS to the new thread. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread. If the thread impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's TokenDefaultDacl owner or members. For more information, see Thread Security and Access Rights.

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the **`ExitThread`** function. Use the **`GetExitCodeThread`** function to get the thread's return value.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **`GetThreadPriority`** and **`SetThreadPriority`** functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **`CloseHandle`**.

The **ExitProcess, ExitThread, CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.

Only one thread in a process can be in a DLL initialization or detach routine at a time.

ExitProcess does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CloseHandle, CreateProcess, CreateRemoteThread, ExitProcess, ExitThread, GetExitCodeThread, GetThreadPriority, ResumeThread, SetThreadPriority, SECURITY_ATTRIBUTES, ThreadProc

---

1.43 CreateToolhelp32Snapshot

Takes a snapshot of the processes and the heaps, modules, and threads used by the processes.

```
CreateToolhelp32Snapshot: procedure
(
    dwFlags:        dword;
    th32ProcessID:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateToolhelp32Snapshot@8" );
```

Parameters

dwFlags

[in] Specifies portions of the system to include in the snapshot. This parameter can be one of the following values.

Value	Meaning
TH32CS_INHERIT	Indicates that the snapshot handle is to be inheritable.
TH32CS_SNAPALL	Equivalent to specifying TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS, and TH32CS_SNAPTHREAD.
TH32CS_SNAPHEAPLIST	Includes the heap list of the specified process in the snapshot.
TH32CS_SNAPMODULE	Includes the module list of the specified process in the snapshot.

TH32CS_SNAPPROCESS	Includes the process list in the snapshot.
TH32CS_SNAPTHREAD	Includes the thread list in the snapshot.

th32ProcessID

[in] Specifies the process identifier. This parameter can be zero to indicate the current process. This parameter is used when the TH32CS_SNAPHEAPLIST or TH32CS_SNAPMODULE value is specified. Otherwise, it is ignored.

Return Values

Returns an open handle to the specified snapshot if successful or – 1 otherwise.

Remarks

The snapshot taken by this function is examined by the other tool help functions to provide their results. Access to the snapshot is read only. The snapshot handle acts like an object handle and is subject to the same rules regarding which processes and threads it is valid in.

To retrieve an extended error status code generated by this function, use the **GetLastError** function.

To destroy the snapshot, use the **CloseHandle** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Tlhelp32.h.
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.44 CreateWaitableTimer

The **CreateWaitableTimer** function creates or opens a waitable timer object.

```
CreateWaitableTimer: procedure
(
    var lpTimerAttributes:  SECURITY_ATTRIBUTES;
        bManualReset:       boolean;
        lpTimerName:        string
);
    stdcall;
    returns( "eax" );
    external( "__imp__CreateWaitableTimerA@12" );
```

Parameters

lpTimerAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new timer object and determines whether child processes can inherit the returned handle. If *lpTimerAttributes* is NULL, the timer object gets a default security descriptor and the handle cannot be inherited.

bManualReset

[in] Specifies the timer type. If *bManualReset* is TRUE, the timer is a manual-reset notification timer. Otherwise, the timer is a synchronization timer.

lpTimerName

[in] Pointer to a null-terminated string specifying the name of the timer object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If the string specified in the *lpTimerName* parameter matches the name of an existing named timer object, the call returns successfully and the **GetLastError** function returns ERROR_ALREADY_EXISTS.

If *lpTimerName* is NULL, the timer object is created without a name.

If *lpTimerName* matches the name of an existing event, semaphore, mutex, job, or file-mapping object, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the timer object. If the named timer object exists before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS.

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

Remarks

The handle returned by **CreateWaitableTimer** is created with the TIMER_ALL_ACCESS access right. This handle can be used in any function that requires a handle to a timer object.

Any thread of the calling process can specify the timer object handle in a call to one of the wait functions.

Multiple processes can have handles to the same timer object, enabling use of the object for interprocess synchronization.

A process created by the **CreateProcess** function can inherit a handle to a timer object if the *lpTimerAttributes* parameter of **CreateWaitableTimer** enables inheritance.

A process can specify the timer object handle in a call to the **DuplicateHandle** function. The resulting handle can be used by another process.

A process can specify the name of a timer object in a call to the **OpenWaitableTimer** or **CreateWaitableTimer** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The timer object is destroyed when its last handle has been closed.

Example

For an example that uses **CreateWaitableTimer**, see Using Waitable Timer Objects.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Synchronization Overview, Synchronization Functions, CancelWaitableTimer, CloseHandle, CreateProcess, Dupli-

1.45 DebugActiveProcess

The **DebugActiveProcess** function enables a debugger to attach to an active process and debug it. To stop debugging the process, you must exit the process. Exiting the debugger will also exit the process.

```
DebugActiveProcess: procedure
(
        dwProcessID:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__DebugActiveProcess@4" );
```

Parameters

dwProcessId

[in] Specifies the identifier for the process to be debugged. The debugger gets debugging access to the process as if it created the process with the DEBUG_ONLY_THIS_PROCESS flag. See the Remarks section for more details.

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 debugger must have appropriate access to the target process; it must be able to open the process for PROCESS_ALL_ACCESS access. On Windows 95/98, the debugger has appropriate access if the process identifier is valid. However, on Windows NT/Windows 2000, **DebugActiveProcess** can fail if the target process was created with a security descriptor that grants the debugger anything less than full access. Note that if the debugging process has the SE_DEBUG_NAME privilege granted and enabled, it can debug any process.

After the system checks the process identifier and determines that a valid debugging attachment is being made, the function returns TRUE. The debugger is then expected to wait for debugging events by using the **WaitForDebugEvent** function. The system suspends all threads in the process and sends the debugger events representing the current state of the process.

The system sends the debugger a single CREATE_PROCESS_DEBUG_EVENT debugging event representing the process specified by the *dwProcessId* parameter. The **lpStartAddress** member of the **CREATE_PROCESS_DEBUG_INFO** structure is NULL.

For each thread currently part of the process, the system sends a CREATE_THREAD_DEBUG_EVENT debugging event. The **lpStartAddress** member of the **CREATE_THREAD_DEBUG_INFO** structure is NULL.

For each dynamic-link library (DLL) currently loaded into the address space of the target process, the system sends a LOAD_DLL_DEBUG_EVENT debugging event. The system arranges for the first thread in the process to execute a breakpoint instruction after it resumes. Continuing this thread causes it to return to whatever it was doing before the debugger was attached.

After all of this has been done, the system resumes all threads in the process. When the first thread in the process resumes, it executes a breakpoint instruction that causes an EXCEPTION_DEBUG_EVENT debugging event to be sent to the debugger. All future debugging events are sent to the debugger by using the normal mechanism and rules.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, CreateProcess, CREATE_PROCESS_DEBUG_INFO, CREATE_THREAD_DEBUG_INFO, WaitForDebugEvent

---

1.46 DebugBreak

The **DebugBreak** function causes a breakpoint exception to occur in the current process so that the calling thread can signal the debugger and force it to take some action. If the process is not being debugged, the search logic of a standard exception handler is used. In most cases, this causes the calling process to terminate because of an unhandled breakpoint exception.

```
DebugBreak: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__DebugBreak@0" );
```

Parameters

This function has no parameters.

Return Values

This function does not return a value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, DebugActiveProcess

1.47 DefineDosDevice

The **DefineDosDevice** function defines, redefines, or deletes MS-DOS device names.

```
DefineDosDevice: procedure
(
    dwFlags:        dword;
    lpDeviceName:   string;
    lpTargetPath:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__DefineDosDeviceA@12" );
```

Parameters

dwFlags

[in] Specifies several controllable aspects of the **DefineDosDevice** function. This parameter can be one or more of the following values.

Value	Meaning
DDD_RAW_TARGET_PATH	If this value is specified, the function does not convert the *lpTargetPath* string from an MS-DOS path to a path, but takes it as is.
DDD_REMOVE_DEFINITION	If this value is specified, the function removes the specified definition for the specified device. To determine which definition to remove, the function walks the list of mappings for the device, looking for a match of *lpTargetPath* against a prefix of each mapping associated with this device. The first mapping that matches is the one removed, and then the function returns. If *lpTargetPath* is NULL or a pointer to a NULL string, the function will remove the first mapping associated with the device and pop the most recent one pushed. If there is nothing left to pop, the device name will be removed. If this value is NOT specified, the string pointed to by the *lpTargetPath* parameter will become the new mapping for this device.
DDD_EXACT_MATCH_ON_REMOVE	If this value is specified along with DDD_REMOVE_DEFINITION, the function will use an exact match to determine which mapping to remove. Use this value to insure that you do not delete something that you did not define.

lpDeviceName

[in] Pointer to an MS-DOS device name string specifying the device the function is defining, redefining, or deleting. The device name string must not have a trailing colon, unless a drive letter (C or D, for example) is being defined, redefined, or deleted. In no case is a trailing backslash allowed.

lpTargetPath

[in] Pointer to a path string that will implement this device. The string is an MS-DOS path string unless the DDD_RAW_TARGET_PATH flag is specified, in which case this string is a path string.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

MS-DOS device names are stored as symbolic links in the object name space. The code that converts an MS-DOS path into a corresponding path uses these symbolic links to map MS-DOS devices and drive letters. The **DefineDosDevice** function provides a mechanism whereby an application can modify the symbolic links used to implement the MS-DOS device name space.

To retrieve the current mapping for a particular MS-DOS device name or to obtain a list of all MS-DOS devices known to the system, use the **QueryDosDevice** function.

MS-DOS Device names are global. After it is defined, an MS-DOS device name remains visible to all processes until either it is explicitly removed or the system restarts.

Windows 2000: To define a drive letter assignment that is persistent across boots and not a network share, use the **SetVolumeMountPoint** function. If the volume to be mounted already has a drive letter assigned to it, use the **DeleteVolumeMountPoint** function to remove the assignment.

Note Drive letters and device names defined at system boot time are protected from redefinition and deletion unless the user is an administrator.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, DeleteVolumeMountPoint, QueryDosDevice, SetVolumeMountPoint

---

1.48 DeleteAtom

The **DeleteAtom** function decrements the reference count of a local string atom. If the atom's reference count is reduced to zero, **DeleteAtom** removes the string associated with the atom from the local atom table.

```
DeleteFiber: procedure
(
    nAtom:  ATOM
);
    stdcall;
    returns( "eax" );
    external( "__imp__DeleteFiber@4" );
```

Parameters

nAtom

[in] Identifies the atom to be deleted.

Return Values

If the function succeeds, the return value is zero.

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

Remarks

A string atom's reference count specifies the number of times the atom has been added to the atom table. The **AddAtom** function increments the count on each call. The **DeleteAtom** function decrements the count on each call but removes the string only if the atom's reference count is zero.

Each call to **AddAtom** should have a corresponding call to **DeleteAtom**. Do not call **DeleteAtom** more times than you call **AddAtom**, or you may delete the atom while other clients are using it.

The **DeleteAtom** function has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, FindAtom, GlobalAddAtom, GlobalDeleteAtom, GlobalFindAtom, MAKEINTATOM

1.49 DeleteFile

The **DeleteFile** function deletes an existing file.

```
DeleteFile: procedure
(
    lpFileName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__DeleteFileA@4" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the file to be deleted.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

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.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

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/2000: 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.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, File I/O Functions, CloseHandle, CreateFile

1.50 DeviceIoControl

The **DeviceIoControl** function sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation.

```
DeviceIoControl: procedure
(
        hDevice:            dword;
        dwIoControlCode:    dword;
    var lpInBuffer:         var;
        nInBufferSize:      dword;
    var lpOutBuffer:        var;
        nOutBufferSize:     dword;
    var lpBytesReturned:    dword;
    var lpOverlapped:       OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__DeviceIoControl@32" );
```

Parameters

hDevice

[in] Handle to the device on which to perform the operation, typically a volume, directory, file, or alternate stream. To retrieve a device handle, use the **`CreateFile`** function.

dwIoControlCode

[in] Specifies the control code for the operation. This value identifies the specific operation to be performed and the type of device on which to perform it.

For a list of the control codes and a short description of each control code, see Device Input and Output Control Codes .

For more detailed information on each control code, see its documentation. In particular, the documentation provides details on the usage of the *lpInBuffer*, *nInBufferSize*, *lpOutBuffer*, *nOutBufferSize*, and *lpBytesReturned* parameters.

lpInBuffer

[in] Pointer to a buffer that contains the data required to perform the operation.

This parameter can be NULL if the *dwIoControlCode* parameter specifies an operation that does not require input data.

nInBufferSize

[in] Specifies the size, in bytes, of the buffer pointed to by *lpInBuffer*.

lpOutBuffer

[out] Pointer to a buffer that receives the operation's output data.

This parameter can be NULL if the *dwIoControlCode* parameter specifies an operation that does not produce output data.

nOutBufferSize

[in] Specifies the size, in bytes, of the buffer pointed to by *lpOutBuffer*.

lpBytesReturned

[out] Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by *lpOutBuffer*.

If the output buffer is too small to return any data, then the call fails, **GetLastError** returns the error code ERROR_INSUFFICIENT_BUFFER, and the returned byte count is zero.

If the output buffer is too small to hold all of the data but can hold some entries, then the operating system returns as much as fits, the call fails, **GetLastError** returns the error code ERROR_MORE_DATA, and *lpBytesReturned* indicates the amount of data returned. Your application should call **DeviceIoControl** again with the same operation, specifying a new starting point.

If *lpOverlapped* is NULL, *lpBytesReturned* cannot be NULL. Even when an operation produces no output data, and *lpOutBuffer* can be NULL, **DeviceIoControl** makes use of the variable pointed to by *lpBytesReturned*. After such an operation, the value of the variable is without meaning.

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

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure.

If *hDevice* was opened with the FILE_FLAG_OVERLAPPED flag, *lpOverlapped* must point to a valid **OVERLAPPED** structure. In this case, the operation is performed as an overlapped (asynchronous) operation. If the device was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the function fails in unpredictable ways.

If *hDevice* was opened without specifying the FILE_FLAG_OVERLAPPED flag, *lpOverlapped* is ignored and **DeviceIoControl** does not return until the operation has been completed, or an error occurs.

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 *hDevice* was opened with FILE_FLAG_OVERLAPPED and the *lpOverlapped* parameter points to an **OVERLAPPED** structure, the operation is performed as an overlapped (asynchronous) operation. In this case, the **OVERLAPPED** structure must contain a handle to a manual-reset event object created by a call to the **CreateEvent** function. For more information on manual-reset event objects, see Synchronization.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Device Input and Output Overview, Device Input and Output Functions, CreateEvent, CreateFile, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED

1.51 DisableThreadLibraryCalls

The **DisableThreadLibraryCalls** function disables the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications for the specified dynamic-link library (DLL). This can reduce the size of the working code set for some applications.

```
DisableThreadLibraryCalls: procedure
(
    hModule:    dword
);
    stdcall;
```

```
    returns( "eax" );
    external( "__imp__DisableThreadLibraryCalls@4" );
```

Parameters

hModule

[in] Handle to the DLL module for which the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications are to be disabled. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The **DisableThreadLibraryCalls** function fails if the DLL specified by *hModule* has active static thread local storage, or if *hModule* is an invalid module handle. To get extended error information, call **GetLastError**.

Remarks

The **DisableThreadLibraryCalls** function lets a DLL disable the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls. This can be a useful optimization for multithreaded applications that have many DLLs, frequently create and delete threads, and whose DLLs do not need these thread-level notifications of attachment/detachment. A remote procedure call (RPC) server application is an example of such an application. In these sorts of applications, DLL initialization routines often remain in memory to service DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications. By disabling the notifications, the DLL initialization code is not paged in because a thread is created or deleted, thus reducing the size of the application's working code set. To implement the optimization, modify a DLL's DLL_PROCESS_ATTACH code to call **DisableThreadLibraryCalls**.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, FreeLibraryAndExitThread

---

1.52 DisconnectNamedPipe

The **DisconnectNamedPipe** function disconnects the server end of a named pipe instance from a client process.

```
DisconnectNamedPipe: procedure
(
    hNamedPipe: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__DisconnectNamedPipe@4" );
```

Parameters

hNamedPipe

[in] Handle to an instance of a named pipe. This handle must be created by the **CreateNamedPipe** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the client end of the named pipe is open, the **DisconnectNamedPipe** function forces that end of the named pipe closed. The client receives an error the next time it attempts to access the pipe. A client that is forced off a pipe by **DisconnectNamedPipe** must still use the **CloseHandle** function to close its end of the pipe.

When the server process disconnects a pipe instance, any unread data in the pipe is discarded. Before disconnecting, the server can make sure data is not lost by calling the **FlushFileBuffers** function, which does not return until the client process has read all the data.

The server process must call **DisconnectNamedPipe** to disconnect a pipe handle from its previous client before the handle can be connected to another client by using the **ConnectNamedPipe** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CloseHandle, ConnectNamedPipe, CreateNamedPipe, FlushFileBuffers

---

1.53 DosDateTimeToFileTime

The **DosDateTimeToFileTime** function converts MS-DOS date and time values to a 64-bit file time.

```
DosDateTimeToFileTime: procedure
(
        wFateDate:  word;
        wFatTime:   word;
    var lpFileTime: FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__DosDateTimeToFileTime@12" );
```

Parameters

wFatDate

[in] Specifies the MS-DOS date. The date is a packed 16-bit value with the following format.

Bits	Contents
0–4	Day of the month (1–31)
5–8	Month (1 = January, 2 = February, and so on)
9–15	Year offset from 1980 (add 1980 to get actual year)

wFatTime

[in] Specifies the MS-DOS time. The time is a packed 16-bit value with the following format.

Bits	Contents
0–4	Second divided by 2
5–10	Minute (0–59)
11–15	Hour (0–23 on a 24-hour clock)

lpFileTime

[out] Pointer to a **FILETIME** structure to receive the converted 64-bit file time.

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

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, FileTimeToDosDateTime, FileTimeToSystemTime, SystemTimeToFileTime

---

1.54 DuplicateHandle

The **DuplicateHandle** function duplicates an object handle. The duplicate handle refers to the same object as the original handle. Therefore, any changes to the object are reflected through both handles. For example, the current file mark for a file handle is always the same for both handles.

```
DuplicateHandle: procedure
(
        hSourceProcessHandle:   dword;
        hSourceHandle:          dword;
        hTargetProcessHandle:   dword;
    var lpTargetProcesHandle:   dword;
        dwDesiredAccess:        dword;
        bInheritHandle:         boolean;
        dwOptions:              dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__DuplicateHandle@28" );
```

Parameters

hSourceProcessHandle

[in] Handle to the process with the handle to duplicate.

Windows NT/2000: The handle must have PROCESS_DUP_HANDLE access. For more information, see Pro-

cess Security and Access Rights.

hSourceHandle

[in] Handle to duplicate. This is an open object handle that is valid in the context of the source process. For a list of objects whose handles can be duplicated, see the following Remarks section.

hTargetProcessHandle

[in] Handle to the process that is to receive the duplicated handle. The handle must have PROCESS_DUP_HANDLE access.

lpTargetHandle

[out] Pointer to a variable that receives the value of the duplicate handle. This handle value is valid in the context of the target process.

If *lpTargetHandle* is NULL, the function duplicates the handle, but does not return the duplicate handle value to the caller. This behavior exists only for backward compatibility with previous versions of this function. You should not use this feature, as you will lose system resources until the target process terminates.

dwDesiredAccess

[in] Specifies the access requested for the new handle. This parameter is ignored if the *dwOptions* parameter specifies the DUPLICATE_SAME_ACCESS flag. Otherwise, the flags that can be specified depend on the type of object whose handle is being duplicated. For the flags that can be specified for each object type, see the following Remarks section. Note that the new handle can have more access than the original handle.

bInheritHandle

[in] Indicates whether the handle is inheritable. If TRUE, the duplicate handle can be inherited by new processes created by the target process. If FALSE, the new handle cannot be inherited.

dwOptions

[in] Specifies optional actions. This parameter can be zero, or any combination of the following values.

Value	Meaning
DUPLICATE_CLOSE_SOURCE	Closes the source handle. This occurs regardless of any error status returned.
DUPLICATE_SAME_ACCESS	Ignores the *dwDesiredAccess* parameter. The duplicate handle has the same access as the source handle.

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

DuplicateHandle can be called by either the source process or the target process. It can also be invoked where the source and target process are the same. For example, a process can use **DuplicateHandle** to create a noninheritable duplicate of an inheritable handle, or a handle with different access than the original handle.

The duplicating process uses the **`GetCurrentProcess`** function to get a handle of itself. To get the other process handle, it may be necessary to use some form of interprocess communication (for example, named pipe or shared memory) to communicate the process identifier to the duplicating process. This identifier is then used in the **`OpenProcess`** function to open a handle.

If the process that calls **DuplicateHandle** is not the target process, the duplicating process must use interprocess communication to pass the value of the duplicate handle to the target process.

DuplicateHandle can duplicate handles to the following types of objects.

Object	Description
Access token	The handle is returned by the **CreateRestrictedToken, DuplicateToken, DuplicateTokenEx, OpenProcessToken,** or **OpenThreadToken** function.
Communications device	The handle is returned by the **CreateFile** function.
Console input	The handle is returned by the **CreateFile** function when CONIN$ is specified, or by the **GetStdHandle** function when STD_INPUT_HANDLE is specified. Console handles can be duplicated for use only in the same process.
Console screen buffer	The handle is returned by the **CreateFile** function when CONOUT$ is specified, or by the **GetStdHandle** function when STD_OUTPUT_HANDLE is specified. Console handles can be duplicated for use only in the same process.
Desktop	The handle is returned by the **GetThreadDesktop** function.
Directory	The handle is returned by the **CreateDirectory** function.
Event	The handle is returned by the **CreateEvent** or **OpenEvent** function.
File	The handle is returned by the **CreateFile** function.
File mapping	The handle is returned by the **CreateFileMapping** function.
Job	The handle is returned by the **CreateJobObject** function.
Mailslot	The handle is returned by the **CreateMailslot** function.
Mutex	The handle is returned by the **CreateMutex** or **OpenMutex** function.
Pipe	A named pipe handle is returned by the **CreateNamedPipe** or **CreateFile** function. An anonymous pipe handle is returned by the **CreatePipe** function.
Process	The handle is returned by the **CreateProcess, GetCurrentProcess,** or **OpenProcess** function.
Registry key	**Windows NT/2000:** The handle is returned by the **RegCreateKey, RegCreateKeyEx, RegOpenKey,** or **RegOpenKeyEx** function. Note that registry key handles returned by the **RegConnectRegistry** function cannot be used in a call to **DuplicateHandle**. **Windows 95/98:** You cannot use **DuplicateHandle** to duplicate registry key handles.
Semaphore	The handle is returned by the **CreateSemaphore** or **OpenSemaphore** function.
Socket	The handle is returned by the **socket** or **accept** function.
Thread	The handle is returned by the **CreateProcess, CreateThread, CreateRemoteThread,** or **GetCurrentThread** function
Timer	The handle is returned by the **CreateWaitableTimer** or **OpenWaitableTimer** function.
Window station	The handle is returned by the **GetProcessWindowStation** function.

Note that **DuplicateHandle** should not be used to duplicate handles to I/O completion ports. In this case, no error is returned, but the duplicate handle cannot be used.

In addition to STANDARD_RIGHTS_REQUIRED, the following access flags can be specified in the *dwDesiredAccess* parameter for the different object types. Note that the new handle can have more access than the original handle. However, in some cases **DuplicateHandle** cannot create a duplicate handle with more access permission than the original handle. For example, a file handle created with GENERIC_READ access cannot be duplicated so that it has both GENERIC_READ and GENERIC_WRITE access.

Any combination of the following access flags is valid for handles to communications devices, console input, console screen buffers, files, and pipes.

Access	Description
GENERIC_READ	Enables read access.
GENERIC_WRITE	Enables write access.

Any combination of the following access flags is valid for file-mapping objects.

Access	Description
FILE_MAP_ALL_ACCESS	Specifies all possible access flags for the file-mapping object.
FILE_MAP_READ	Enables mapping the object into memory that permits read access.
FILE_MAP_WRITE	Enables mapping the object into memory that permits write access. For write access, PAGE_READWRITE protection must have been specified when the file-mapping object was created by the **`CreateFileMapping`** function.

Any combination of the following flags is valid for mutex objects.

Access	Description
MUTEX_ALL_ACCESS	Specifies all possible access flags for the mutex object.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the mutex handle in any of the wait functions to acquire ownership of the mutex, or in the **`ReleaseMutex`** function to release ownership.

Any combination of the following access flags is valid for semaphore objects.

Access	Description
SEMAPHORE_ALL_ACCESS	Specifies all possible access flags for the semaphore object.
SEMAPHORE_MODIFY_STATE	Enables use of the semaphore handle in the **`ReleaseSemaphore`** function to modify the semaphore's count.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the semaphore handle in any of the wait functions to wait for the semaphore's state to be signaled.

Any combination of the following access flags is valid for event objects.

Access	Description
EVENT_ALL_ACCESS	Specifies all possible access flags for the event object.

EVENT_MODIFY_STATE	Enables use of the event handle in the **SetEvent** and **ResetEvent** functions to modify the event's state.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the event handle in any of the wait functions to wait for the event's state to be signaled.

Any combination of the following access flags is valid for handles to registry keys.

Value	Meaning
KEY_ALL_ACCESS	Specifies all possible flags for the registry key.
KEY_CREATE_LINK	Enables using the handle to create a link to a registry-key object.
KEY_CREATE_SUB_KEY	Enables using the handle to create a subkey of a registry-key object.
KEY_ENUMERATE_SUB_KEYS	Enables using the handle to enumerate the subkeys of a registry-key object.
KEY_EXECUTE	Equivalent to KEY_READ.
KEY_NOTIFY	Enables using the handle to request change notifications for a registry key or for subkeys of a registry key.
KEY_QUERY_VALUE	Enables using the handle to query a value of a registry-key object.
KEY_READ	Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.
KEY_SET_VALUE	Enables using the handle to create or set a value of a registry-key object.
KEY_WRITE	Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY values.

Any combination of the following access flags is valid for process objects.

Access	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the **CreateRemoteThread** function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the **DuplicateHandle** function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the **GetExitCodeProcess** and **GetPriorityClass** functions to read information from the process object.
PROCESS_SET_INFORMATION	Enables using the process handle in the **SetPriorityClass** function to set the process's priority class.
PROCESS_TERMINATE	Enables using the process handle in the **TerminateProcess** function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the **VirtualProtectEx** and **WriteProcessMemory** functions to modify the virtual memory of the process.

PROCESS_VM_READ	Enables using the process handle in the **ReadProcessMemory** function to read from the virtual memory of the process.
PROCESS_VM_WRITE	Enables using the process handle in the **WriteProcessMemory** function to write to the virtual memory of the process.
SYNCHRONIZE	**Windows NT/2000:** Enables using the process handle in any of the wait functions to wait for the process to terminate.

Any combination of the following access flags is valid for thread objects.

Access	Description
SYNCHRONIZE	**Windows NT/2000:** Enables using the thread handle in any of the wait functions to wait for the thread to terminate.
THREAD_ALL_ACCESS	Specifies all possible access flags for the thread object.
THREAD_DIRECT_IMPERSONATION	Used internally.
THREAD_GET_CONTEXT	Enables using the thread handle in the **GetThreadContext** function to read the thread's context.
THREAD_IMPERSONATE	Used internally.
THREAD_QUERY_INFORMATION	Enables using the thread handle in the **GetExitCodeThread, GetThreadPriority,** and **GetThreadSelectorEntry** functions to read information from the thread object.
THREAD_SET_CONTEXT	Enables using the thread handle in the **SetThreadContext** function to set the thread's context.
THREAD_SET_INFORMATION	Enables using the thread handle in the **SetThreadPriority** function to set the thread's priority.
THREAD_SET_THREAD_TOKEN	Used internally.
THREAD_SUSPEND_RESUME	Enables using the thread handle in the **SuspendThread** or **ResumeThread** functions to suspend or resume a thread.
THREAD_TERMINATE	Enables using the thread handle in the **TerminateThread** function to terminate the thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Handles and Objects Overview, Handle and Object Functions, CloseHandle

---

1.55 EndUpdateResource

The **EndUpdateResource** function ends a resource update in an executable file.

```
EndUpdateResource: procedure
(
    hUpdate:    dword;
    fDiscard:   boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__EndUpdateResourceA@8" );
```

Parameters

hUpdate

[in] Handle used in a resource update. This handle is returned by the **BeginUpdateResource** function.

fDiscard

[in] Specifies whether to write resource updates to an executable file. If this parameter is TRUE, no changes are made to the executable file. If it is FALSE, the changes are made.

Return Values

If the function succeeds and the accumulated resource modifications specified by calls to the **UpdateResource** function are written to the specified executable file, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, BeginUpdateResource, UpdateResource

---

1.56 EnterCriticalSection

The **EnterCriticalSection** function waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership.

```
EnterCriticalSection: procedure
(
    lpCriticalSection:  CRITICAL_SECTION
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnterCriticalSection@4" );
```

Parameters

lpCriticalSection

[in/out] Pointer to the critical section object.

Return Values

This function does not return a value.

In low memory situations, **EnterCriticalSection** can raise a STATUS_INVALID_HANDLE exception. To avoid

problems, use structured exception handling, or call the **InitializeCriticalSectionAndSpinCount** function to preallocate the event used by **EnterCriticalSection** instead of calling the **InitializeCriticalSection** function, which forces **EnterCriticalSection** to allocate the event.

Remarks

The threads of a single process can use a critical section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** to initialize the object.

To enable mutually exclusive access to a shared resource, each thread calls the **EnterCriticalSection** or **TryEnterCriticalSection** function to request ownership of the critical section before executing any section of code that accesses the protected resource. The difference is that **TryEnterCriticalSection** returns immediately, regardless of whether it obtained ownership of the critical section, while **EnterCriticalSection** blocks until the thread can take ownership of the critical section. When it has finished executing the protected code, the thread uses the **LeaveCriticalSection** function to relinquish ownership, enabling another thread to become owner and access the protected resource. The thread must call **LeaveCriticalSection** once for each time that it entered the critical section. The thread enters the critical section each time **EnterCriticalSection** and **TryEnterCriticalSection** succeed.

After a thread has ownership of a critical section, it can make additional calls to **EnterCriticalSection** or **TryEnterCriticalSection** without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.

If a critical section is deleted while it is still owned, the state of the threads waiting for ownership of the deleted critical section is undefined.

Example

For an example that uses **EnterCriticalSection**, see Using Critical Section Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, DeleteCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection TryEnterCriticalSection

---

1.57 EnumCalendarInfo

The **EnumCalendarInfo** function enumerates calendar information for a specified locale. The *CalType* parameter specifies the type of calendar information to enumerate. The function returns the specified calendar information for all applicable calendars for the locale or, for a single requested calendar, depending on the value of the *Calendar* parameter.

The **EnumCalendarInfo** function enumerates the calendar information by calling an application defined–callback function. It passes the callback function a pointer to a buffer containing the requested calendar information. This continues until either the last applicable calendar is found or the callback function returns FALSE.

To receive a calendar identifier in addition to the calendar information provided by **EnumCalendarInfo**, use the

EnumCalendarInfoEx function.

```
EnumCalendarInfo: procedure
(
    pCalInfoEnumProc:   CALINFO_ENUMPROC;
    Locale:             LCID;
    Calendar:           CALID;
    CalType:            CALTYPE
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumCalendarInfoA@16" );
```

Parameters

pCalInfoEnumProc

[in] Pointer to an application defined—callback function. For more information, see the **EnumCalendarInfoProc** callback function.

Locale

[in] Specifies the locale for which to retrieve calendar information. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

Calendar

[in] Specifies the calendar for which information is requested. The following values are defined.

Value	Meaning
ENUM_ALL_CALENDARS	All applicable calendars for the specified locale.
CAL_GREGORIAN	Gregorian (localized).
CAL_GREGORIAN_US	Gregorian (English strings always).
CAL_JAPAN	Japanese Emperor Era.
CAL_TAIWAN	Taiwan Era.
CAL_KOREA	Korean Tangun Era.
CAL_HIJRI	Hijri (Arabic Lunar).
CAL_THAI	Thai.
CAL_HEBREW	Hebrew (Lunar).
CAL_GREGORIAN_ME_FRENCH	Gregorian Middle East French.
CAL_GREGORIAN_ARABIC	Gregorian Arabic.
CAL_GREGORIAN_XLIT_ENGLISH	Gregorian transliterated English.
CAL_GREGORIAN_XLIT_FRENCH	Gregorian transliterated French.

CalType

[in] Specifies the type of calendar information to be returned. See the listing of constant values in Calendar Type Information. Note that only one **CALTYPE** value can be specified per call of this function, except where noted.

Windows 98: EnumCalendarInfo does not support CAL_ITWODIGITYEARMAX.

Windows Me: EnumCalendarInfo supports CAL_ITWODIGITYEARMAX.

Return Values

If the function succeeds, the return values is a nonzero value.

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_BADDB

ERROR_INVALID_FLAGS

ERROR_INVALID_PARAMETER

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumCalendarInfoEx, EnumCalendarInfoProc, EnumDateFormats, MAKELCID

1.58 EnumCalendarInfoEx

The **EnumCalendarInfoEx** function enumerates calendar information for a specified locale. The *CalType* parameter specifies the type of calendar information to enumerate. The function returns the specified calendar information for all applicable calendars for the locale or, for a single requested calendar, depending on the value of the *Calendar* parameter.

The **EnumCalendarInfoEx** function enumerates the calendar information by calling an application defined–callback function. It passes the callback function a pointer to a buffer containing the requested calendar information and a calendar identifier (CALID). This continues until either the last applicable calendar is found or the callback function returns FALSE.

```
EnumCalendarInfoEx: procedure
(
     pCalInfoEnumProcEx:CALINFO_ENUMPROCEX;
     Locale:            LCID;
     Calendar:     CALID;
     CalendarType:CALTYPE
);
     stdcall;
     returns( "eax" );
     external( "__imp__EnumCalendarInfoExA@16" );
```

Parameters

pCalInfoEnumProcEx

[in] Pointer to an application defined—callback function. For more information, see the **EnumCalendarInfoPro-**

cEx callback function.

Locale

[in] Specifies the locale for which to retrieve calendar information. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

Calendar

[in] Specifies the calendar for which information is requested. The following values are defined.

Value	Meaning
ENUM_ALL_CALENDARS	All applicable calendars for the specified locale.
CAL_GREGORIAN	Gregorian (localized)
CAL_GREGORIAN_US	Gregorian (English strings always)
CAL_JAPAN	Japanese Emperor Era
CAL_TAIWAN	Taiwan Era
CAL_KOREA	Korean Tangun Era
CAL_HIJRI	Hijri (Arabic Lunar)
CAL_THAI	Thai
CAL_HEBREW	Hebrew (Lunar)
CAL_GREGORIAN_ME_FRENCH	Gregorian Middle East French
CAL_GREGORIAN_ARABIC	Gregorian Arabic
CAL_GREGORIAN_XLIT_ENGLISH	Gregorian transliterated English
CAL_GREGORIAN_XLIT_FRENCH	Gregorian transliterated French

CalType

[in] Specifies the type of calendar information to be returned. See the listing of constant values in Calendar Type Information. Note that only one **CALTYPE** value can be specified per call of this function, except where noted.

Return Values

If the function succeeds, the return values is a nonzero value.

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_BADDB

ERROR_INVALID_FLAGS

ERROR_INVALID_PARAMETER

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumCalendarInfoEx, EnumCalendarInfoProc, EnumDateFormats

---

1.59 EnumDateFormats

The **EnumDateFormats** function enumerates the long or short date formats that are available for a specified locale. The value of the *dwFlags* parameter determines whether the long or short date formats are enumerated. The function enumerates the date formats by passing date format string pointers, one at a time, to the specified application–defined callback function. This continues until the last date format is found or the callback function returns FALSE.

To receive a calendar identifier in addition to the date format information provided by **EnumDateFormats**, use the **EnumDateFormatsEx** function.

```
EnumDateFormats: procedure
(
    lpDateFmtEnumProc:  DATEFMT_ENUMPROC;
    Locale:             LCID;
    dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumDateFormatsA@12" );
```

Parameters

lpDateFmtEnumProc

[in] Pointer to an application—defined callback function. The **EnumDateFormats** function enumerates date formats by making repeated calls to this callback function. For more information, see the **EnumDateFormatsProc** callback function.

Locale

[in] Specifies the locale for which to retrieve date format information. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Specifies the date formats of interest. Use one of the following values.

Value	Meaning

LOCALE_USE_CP_ACP	Uses the system ANSI code page for string translation instead of the locale's code page.
DATE_SHORTDATE	Return short date formats. This value cannot be used with DATE_LONGDATE.
DATE_LONGDATE	Return long date formats. This value cannot be used with DATE_SHORTDATE.
DATE_YEARMONTH	Return year/month formats.

Return Values

If the function succeeds, the return values is a nonzero value.

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_PARAM

ETERERROR_BADDB

ERROR_INVALID FLAGS

Remarks

This function will return all date formats for the specified locale, including alternate calendars, if any. However, the calendar identifier is not returned along with the date format in the callback function, so formats for locales with alternate calendars are difficult to use. To get the date formats for locales with alternate calendars, use **EnumDateFormatsEx**.

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumDateFormatsEx, EnumDateFormatsProc, EnumCalendarInfo, EnumTimeFormats

---

1.60 EnumDateFormatsEx

The **EnumDateFormatsEx** function enumerates the long or short date formats that are available for a specified locale, including date formats for any alternate calendars. The value of the *dwFlags* parameter determines whether the long date, short date, or year/month formats are enumerated. The function enumerates the date formats by passing date format string pointers, one at a time, to the specified application–defined callback function. This continues until the last date format is found or the callback function returns FALSE.

```
EnumDateFormatsEx: procedure
(
    lpDateFmtEnumProcEx:    DATEFMT_ENUMPROCEX;
    Locale:                 LCID;
    dwFlags:                dword
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__EnumDateFormatsExA@12" );
```

Parameters

lpDateFmtEnumProcEx

[in] Pointer to an application—defined callback function. The **EnumDateFormatsEx** function enumerates date formats by making repeated calls to this callback function. For more information, see the **EnumDateFormatsProcEx** callback function.

Locale

[in] Specifies the locale for which to retrieve date format information. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Specifies the date formats that are of interest. Use one of the following values.

Value	Meaning
LOCALE_USE_CP_ACP	Uses the system ANSI code page for string translation instead of the locale's code page.
DATE_SHORTDATE	Return short date formats. This value cannot be used with DATE_LONGDATE.
DATE_LONGDATE	Return long date formats. This value cannot be used with DATE_SHORTDATE.
DATE_YEARMONTH	Return year/month formats.

Return Values

If the function succeeds, the return values is a nonzero value.

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_PARAM

ETERERROR_BADDB

ERROR_INVALID FLAGS

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in Winnls.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumDateFormatsProcEx, EnumCalendarInfo, EnumTimeFormats

---

1.61 EnumResourceLanguages

The **EnumResourceLanguages** function searches a module for each resource of the specified type and name and passes the language of each resource it locates to a defined callback function.

```
EnumResourceLanguages: procedure
(
     hModule:  dword;
     lpType:   string;
     lpName:   string;
     lpEnumFunc:ENUMRESLANGPROC;
     lParam:   dword
);
     stdcall;
     returns( "eax" );
     external( "__imp__EnumResourceLanguagesA@20" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resources for which the languages are to be enumerated. If this parameter is NULL, the function enumerates the resource languages in the module used to create the current process.

lpType

[in] Pointer to a null-terminated string specifying the type of the resource for which the language is being enumerated. For standard resource types, this parameter can be one of the following values.

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource

RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry
RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource
RT_VXD	VXD

lpName

[in] Pointer to a null-terminated string specifying the name of the resource for which the language is being enumerated.

lpEnumFunc

[in] Pointer to the callback function to be called for each enumerated resource language. For more information, see **EnumResLangProc**.

lParam

[in] Specifies an application-defined value passed to the callback function. This parameter may be used in error checking.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If a resource has an ID, the ID is returned to the callback function; otherwise the resource name is returned to the callback function. For more information, see **EnumResLangProc**.

The **EnumResourceLanguages** function continues to enumerate resource languages until the callback function returns FALSE or all resource languages have been enumerated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, EnumResLangProc, EnumResourceNames, EnumResourceTypes

---

1.62 EnumResourceNames

The **EnumResourceNames** function searches a module for each resource of the specified type and passes either the name or the ID of each resource it locates to an application-defined callback function.

```
EnumResourceNames: procedure
(
     hModule:dword;
     lpszType:string;
     lpEnumFunc:ENUMRESNAMEPROC;
     lParam:   dword
);
```

```
stdcall;
returns( "eax" );
external( "__imp__EnumResourceNamesA@16" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resources that are to be enumerated. If this parameter is NULL, the function enumerates the resources in the module used to create the current process.

lpszType

[in] Pointer to a null-terminated string specifying the type of resource to be enumerated. For standard resource types, this parameter can be one of the following values.

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource
RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry
RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource
RT_VXD	VXD

lpEnumFunc

[in] Pointer to the callback function to be called for each enumerated resource name. For more information, see **EnumResNameProc**.

lParam

[in] Specifies an application-defined value passed to the callback function. This parameter can be used in error checking.

Return Values

If the function succeeds, the return value is nonzero.

If the function does not find a resource of the type specified, or if the function fails for another reason, the return value is zero. To get extended error information, call **GetLastError**.

GetLastError value	Meaning
NO_ERROR	**Windows 95/98:** No resource of the specified type was found.
ERROR_RESOURCE_TYPE_NOT_FOUND	**Windows NT/2000:** No resource of the specified type was found.
any other return value	Some other type of error occurred.

Remarks

If a resource has an ID, the ID is returned to the callback function; otherwise the resource name is returned to the callback function. For more information, see **EnumResNameProc**.

The **EnumResourceNames** function continues to enumerate resources until the callback function returns FALSE or all resources have been enumerated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on all platforms.

See Also

Resources Overview, Resource Functions, EnumResNameProc, EnumResourceLanguages, EnumResourceTypes

---

1.63 EnumResourceTypes

The **EnumResourceTypes** function searches a module for resources and passes each resource type it finds to an application-defined callback function.

```
EnumResourceTypes: procedure
(
    hModule:    dword;
    lpEnumFunc: ENUMRESTYPEPROC;
    lParam:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumResourceTypesA@12" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resources for which the types are to be enumerated. If this parameter is NULL, the function enumerates the resource types in the module used to create the current process.

lpEnumFunc

[in] Pointer to the callback function to be called for each enumerated resource type. For more information, see the **EnumResTypeProc** function.

lParam

[in] Specifies an application-defined value passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **EnumResourceTypes** function continues to enumerate resource types until the callback function returns FALSE or all resource types have been enumerated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on all platforms.

See Also

Resources Overview, Resource Functions, EnumResourceLanguages, EnumResourceNames, EnumResTypeProc

---

1.64 EnumSystemCodePages

The **EnumSystemCodePages** function enumerates the code pages that are either installed on or supported by a system. The *dwFlags* parameter determines whether the function enumerates installed or supported code pages. The function enumerates the code pages by passing code page identifiers, one at a time, to the specified application defined–callback function. This continues until all of the installed or supported code page identifiers have been passed to the callback function, or the callback function returns FALSE.

```
EnumSystemCodePages: procedure
(
    lpCodePageEnumProc: CODEPAGE_ENUMPROC;
    dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumSystemCodePagesA@8" );
```

Parameters

lpCodePageEnumProc

[in] Pointer to an application defined—callback function. The **EnumSystemCodePages** function enumerates code pages by making repeated calls to this callback function. For more information, see the **EnumCodePagesProc** callback function.

dwFlags

[in] Specifies the code pages to enumerate. This parameter can be one of the following values.

Value	Meaning
CP_INSTALLED	Enumerate only installed code pages.

CP_SUPPORTED	Enumerate all supported code pages.

Return Values

If the function succeeds, the return values is a nonzero value.

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_PARAMETER

ERROR_BADDB

ERROR_INVALID FLAGS

Remarks

The CP_INSTALLED and CP_SUPPORTED flags are mutually exclusive.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnls.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

National Language Support Overview, **National Language Support Functions**, **EnumCodePagesProc**

1.65 EnumSystemLocales

The **EnumSystemLocales** function enumerates the locales that are either installed on or supported by a system. The *dwFlags* parameter determines whether the function enumerates installed or supported system locales. The function enumerates locales by passing locale identifiers, one at a time, to the specified application defined–callback function. This continues until all of the installed or supported locale identifiers have been passed to the callback function or the callback function returns FALSE.

```
EnumSystemLocales: procedure
(
    lpLocaleEnumProc:   LOCALE_ENUMPROC;
    dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumSystemLocalesA@8" );
```

Parameters

lpLocaleEnumProc

[in] Pointer to an application defined—callback function. The **EnumSystemLocales** function enumerates locales by making repeated calls to this callback function. For more information, see the **EnumLocalesProc** callback function.

dwFlags

[in] Locale identifiers to enumerate. This parameter can be one or more of the following values.

Value	Meaning

LCID_INSTALLED	Enumerate only installed locale identifiers. This value cannot be used with LCID_SUPPORTED.
LCID_SUPPORTED	Enumerate all supported locale identifiers. This value cannot be used with LCID_INSTALLED.
LCID_ALTERNATE_SORTS	Enumerate only the alternate sorts. If this value is used with either LCID_INSTALLED or LCID_SUPPORTED, then the installed or supported locales will be returned as well as the alternate sort locale IDs.

Return Values

If the function succeeds, the return values is a nonzero value.

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_PARAMETER

ERROR_BADDB

ERROR_INVALID FLAGS

Remarks

The LCID_INSTALLED and LCID_SUPPORTED flags are mutually exclusive.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumLocalesProc

---

1.66 EnumTimeFormats

The **EnumTimeFormats** function enumerates the time formats that are available for a specified locale. The function enumerates the time formats by passing a pointer to a buffer containing a time format to an application defined–callback function. It continues to do so until the last time format is found or the callback function returns FALSE.

```
EnumTimeFormats: procedure
(
    lpTimeFmtEnumProc:  TIMEFMT_ENUMPROC;
    Locale:             LCID;
    dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EnumTimeFormatsA@12" );
```

Parameters

lpTimeFmtEnumProc

[in] Pointer to an application defined—callback function. For more information, see **EnumTimeFormatsProc**.

Locale

[in] Specifies the locale to retrieve time format information for. This parameter can be a locale identifier created by the **MAKELCID** macro, or by one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Currently, only the following value is defined.

Value	Meaning
LOCALE_USE_CP_ACP	Use the system ANSI code page for string translation instead of the locale's code page.

Return Values

If the function succeeds, the return values is a nonzero value.

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_PARAMETER

ERROR_BADDB

ERROR_INVALID FLAGS

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, EnumTimeFormatsProc, EnumCalendarInfo, EnumDateFormats

---

1.67 EraseTape

The **EraseTape** function erases all or part of a tape.

```
EraseTape: procedure
(
    hDevice:        dword;
    dwEraseType:    dword;
    bImmediate:     boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__EraseTape@12" );
```

Parameters

hDevice

[in] Handle to the device where the tape is to be erased. This handle is created by using the **`CreateFile`** function.

dwEraseType

[in] Specifies the erasing technique. This parameter can be one of the following values.

Value	Description
TAPE_ERASE_LONG	Erases the tape from the current position to the end of the current partition.
TAPE_ERASE_SHORT	Writes an erase gap or end-of-data marker at the current position.

bImmediate

[in] Specifies whether to return as soon as the erase operation begins. If this parameter is TRUE, the function returns immediately; if it is FALSE, the function does not return until the erase operation has been completed.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error code	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

Some tape devices do not support certain tape operations. To determine your tape device's capabilities, see your tape device documentation and use the **GetTapeParameters** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, GetTapeParameters

---

1.68 EscapeCommFunction

The **EscapeCommFunction** function directs a specified communications device to perform an extended function.

```
EscapeCommFunction: procedure
(
    hFile:  dword;
    dwFunc: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__EscapeCommFunction@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

dwFunc

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

Value	Meaning
CLRDTR	Clears the DTR (data-terminal-ready) signal.
CLRRTS	Clears the RTS (request-to-send) signal.
SETDTR	Sends the DTR (data-terminal-ready) signal.
SETRTS	Sends the RTS (request-to-send) signal.
SETXOFF	Causes transmission to act as if an XOFF character has been received.
SETXON	Causes transmission to act as if an XON character has been received.
SETBREAK	Suspends character transmission and places the transmission line in a break state until the **ClearCommBreak** function is called (or **EscapeCommFunction** is called with the CLRBREAK extended function code). The SETBREAK extended function code is identical to the **SetCommBreak** function. Note that this extended function does not flush data that has not been transmitted.
CLRBREAK	Restores character transmission and places the transmission line in a nonbreak state. The CLRBREAK extended function code is identical to the **ClearCommBreak** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, ClearCommBreak, CreateFile, SetCommBreak

---

1.69 ExitProcess

The **ExitProcess** function ends a process and all its threads.

```
ExitProcess: procedure
(
    uExitCode:uns32
);
    stdcall;
    returns( "eax" );
    external( "__imp__ExitProcess@4" );
```

Parameters

uExitCode

[in] Specifies the exit code for the process, and for all threads that are terminated as a result of this call. Use the **GetExitCodeProcess** function to retrieve the process's exit value. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

Return Values

This function does not return a value.

Remarks

ExitProcess is the preferred method of ending a process. This function provides a clean process shutdown. This includes calling the entry-point function of all attached dynamic-link libraries (DLLs) with a value indicating that the process is detaching from the DLL. If a process terminates by calling **TerminateProcess**, the DLLs that the process is attached to are not notified of the process termination.

After all attached DLLs have executed any process termination value, this function terminates the current process.

Terminating a process causes the following:

All of the object handles opened by the process are closed.

All of the threads in the process terminate their execution.

The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.

The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.

The termination status of the process changes from STILL_ACTIVE to the exit value of the process.

Terminating a process does not cause child processes to be terminated.

Terminating a process does not necessarily remove the process object from the operating system. A process object is deleted when the last handle to the process is closed.

The **ExitProcess**, **ExitThread, CreateThread, CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.

Only one thread in a process can be in a DLL initialization or detach routine at a time.

If any process is in its DLL initialization or detach routine, **ExitProcess** does not return.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, CreateRemoteThread, CreateThread, ExitThread, GetExitCodeProcess, GetExitCodeThread, OpenProcess, TerminateProcess

---

1.70 ExitThread

The **ExitThread** function ends a thread.

```
ExitThread: procedure
(
    dwExitCode: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ExitThread@4" );
```

Parameters

dwExitCode

[in] Specifies the exit code for the calling thread. Use the **GetExitCodeThread** function to retrieve a thread's exit code.

Return Values

This function does not return a value.

Remarks

ExitThread is the preferred method of exiting a thread. When this function is called (either explicitly or by returning from a thread procedure), the current thread's stack is deallocated and the thread terminates. The entry-point function of all attached dynamic-link libraries (DLLs) is invoked with a value indicating that the thread is detaching from the DLL.

If the thread is the last thread in the process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from STILL_ACTIVE to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the operating system. A thread object is

deleted when the last handle to the thread is closed.

The **ExitProcess**, **ExitThread**, **CreateThread, CreateRemoteThread** functions, and a process that is starting (as the result of a **CreateProcess** call) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.

Only one thread in a process can be in a DLL initialization or detach routine at a time.

ExitProcess does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **_beginthread** and **_endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, CreateRemoteThread, CreateThread, ExitProcess, FreeLibraryAndExitThread, GetExitCodeThread, OpenThread, TerminateThread

---

1.71 ExpandEnvironmentStrings

The **ExpandEnvironmentStrings** function expands environment-variable strings and replaces them with their defined values.

```
ExpandEnvironmentStrings: procedure
(
    lpSrc:  string;
    lpDst:  string;
    nSize:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ExpandEnvironmentStringsA@12" );
```

Parameters

lpSrc

[in] Pointer to a null-terminated string that contains environment-variable strings of the form: %*variableName*%. For each such reference, the **%***variableName***%** portion is replaced with the current value of that environment variable.

The replacement rules are the same as those used by the command interpreter. Case is ignored when looking up the environment-variable name. If the name is not found, the **%***variableName***%** portion is left undisturbed.

lpDst

[out] Pointer to a buffer to receive a copy of the source buffer after all environment-variable name substitutions

have been performed.

nSize

[in] Specifies the maximum number of **TCHARs** that can be stored in the buffer pointed to by the *lpDst* parameter, including the terminating null character.

Return Values

If the function succeeds, the return value is the number of **TCHARs** stored in the destination buffer, including the terminating null character. If the destination buffer is too small to hold the expanded string, the return value is the required buffer size, in **TCHARs**.

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

Remarks

There are certain restrictions on the size of the *lpSrc* and *lpDst* buffers.

On Windows NT 4.0 and earlier, these buffers are limited to 32K.

On Windows 2000, these buffers are limited to 32K in the ANSI version, but there is no size restriction in the Unicode version.

On Windows 95/98, the ANSI version has no size restriction; the Unicode version is not supported.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

System Information Overview, System Information Functions

1.72 FatalAppExit

The **FatalAppExit** function displays a message box and terminates the application when the message box is closed. If the system is running with a kernel debugger, the message box gives the user the opportunity to terminate the application or to cancel the message box and return to the application that called **FatalAppExit**.

```
FatalAppExit: procedure
(
    uAction:        uns32;
    lpMessageText:  string
);
    stdcall;
    returns( "eax" );
    external( "__imp__FatalAppExitA@8" );
```

Parameters

uAction

Reserved; must be zero.

lpMessageText

[in] Pointer to a null-terminated string that is displayed in the message box. The message is displayed on a single

line. To accommodate low-resolution screens, the string should be no more than 35 characters in length.

Return Values

This function does not return a value.

Remarks

An application calls **FatalAppExit** only when it is not capable of terminating any other way. **FatalAppExit** may not always free an application's memory or close its files, and it may cause a general failure of the system. An application that encounters an unexpected error should terminate by freeing all its memory and returning from its main message loop.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Error Handling Overview, Error Handling Functions, FatalExit

1.73 FatalExit

The **FatalExit** function transfers execution control to the debugger. The behavior of the debugger thereafter is specific to the type of debugger used.

```
FatalExit: procedure
(
    ExitCode:   int32
);
    stdcall;
    returns( "eax" );
    external( "__imp__FatalExit@4" );
```

Parameters

ExitCode

[in] Specifies the error code associated with the exit.

Return Values

This function does not return a value.

Remarks

An application should only use **FatalExit** for debugging purposes. It should not call the function in a retail version of the application because doing so will terminate the application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, FatalAppExit

1.74 FileTimeToDosDateTime

The **FileTimeToDosDateTime** function converts a 64-bit file time to MS-DOS date and time values.

```
FileTimeToDosDateTime: procedure
(
    var lpFileTime: FILETIME;
    var lpFatDate:  word;
    var lpFatTime:  word
);
    stdcall;
    returns( "eax" );
    external( "__imp__FileTimeToDosDateTime@12" );
```

Parameters

lpFileTime

[in] Pointer to a **FILETIME** structure containing the 64-bit file time to convert to MS-DOS date and time format.

lpFatDate

[out] Pointer to a variable to receive the MS-DOS date. The date is a packed 16-bit value with the following format.

Bits	Contents
0–4	Day of the month (1–31)
5–8	Month (1 = January, 2 = February, etc.)
9–15	Year offset from 1980 (add 1980 to get actual year)

lpFatTime

[out] Pointer to a variable to receive the MS-DOS time. The time is a packed 16-bit value with the following format.

Bits	Contents
0–4	Second divided by 2
5–10	Minute (0–59)
11–15	Hour (0–23 on a 24-hour clock)

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 MS-DOS date format can represent only dates between 1/1/1980 and 12/31/2107; this conversion fails if the input file time is outside this range.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, DosDateTimeToFileTime, FileTimeToSystemTime, SystemTimeToFileTime

---

1.75 FileTimeToLocalFileTime

The **FileTimeToLocalFileTime** function converts a file time based on the Coordinated Universal Time (UTC) to a local file time.

```
FileTimeToLocalFileTime: procedure
(
    var lpFileTime:         FILETIME;
    var lpLocalFileTime:    FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__FileTimeToLocalFileTime@8" );
```

Parameters

lpFileTime

[in] Pointer to a **FILETIME** structure containing the UTC-based file time to be converted into a local file time.

lpLocalFileTime

[out] Pointer to a **FILETIME** structure to receive the converted local file time. This parameter cannot be the same as the *lpFileTime* parameter.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

FileTimeToLocalFileTime uses the current settings for the time zone and daylight saving time. Therefore, if it is daylight saving time, this function will take daylight saving time into account, even if the time you are converting is in standard time.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, LocalFileTimeToFileTime

1.76 FileTimeToSystemTime

The **FileTimeToSystemTime** function converts a 64-bit file time to system time format.

```
FileTimeToSystemTime: procedure
(
    var lpFileTime:     FILETIME;
    var lpSystemTime:   SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__FileTimeToSystemTime@8" );
```

Parameters

lpFileTime

[in] Pointer to a **FILETIME** structure containing the file time to convert to system date and time format.

The **FileTimeToSystemTime** function only works with **FILETIME** values that are less than 0x8000000000000000. The function fails with values equal to or greater than that.

lpSystemTime

[out] Pointer to a **SYSTEMTIME** structure to receive the converted file time.

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

As noted above, the function fails for **FILETIME** values that are equal to or greater than 0x8000000000000000.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, DosDateTimeToFileTime, FILETIME, FileTimeToDosDateTime, SYSTEMTIME, SystemTimeToFileTime

1.77 FillConsoleOutputAttribute

The **FillConsoleOutputAttribute** function sets the text and background color attributes for a specified number of character cells, beginning at the specified coordinates in a screen buffer.

```
FillConsoleOutputCharacter: procedure
(
    hConsoleOutput:             dword;
    wAttribute:                 word;
```

```
        nLength:                    dword;
        dwWriteCoord:               COORD;
        var lpNumberOfAttrsWritten: dword
    );
        stdcall;
        returns( "eax" );
        external( "__imp__FillConsoleOutputCharacterA@20" );
```

Parameters

hConsoleOutput

[in] Handle to a screen buffer. The handle must have GENERIC_WRITE access.

wAttribute

[in] Specifies the foreground and background color attributes to write to the screen buffer. Any combination of the following values can be specified: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces white text on a black background:

```
FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE
```

nLength

[in] Specifies the number of character cells to be set to the specified color attributes.

dwWriteCoord

[in] Specifies a **COORD** structure containing the screen buffer coordinates of the first cell whose attributes are to be set.

lpNumberOfAttrsWritten

[out] Pointer to the variable that receives the number of character cells whose attributes were actually set.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of character cells whose attributes are to be set extends beyond the end of the specified row in the screen buffer, the cells of the next row are set. If the number of cells to write to extends beyond the end of the screen buffer, the cells are written up to the end of the screen buffer.

The character values at the positions written to are not changed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, COORD, FillConsoleOutputCharacter, SetConsoleTextAttribute, WriteConsoleOutputAttribute

1.78 FindAtom

The **FindAtom** function searches the local atom table for the specified character string and retrieves the atom associated with that string.

```
FindAtom: procedure
(
    lpString:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindAtomA@4" );
```

Parameters

lpString

[in] Pointer to the null-terminated character string to search for.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the atom associated with the given string.

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

Remarks

Even though the system preserves the case of a string in an atom table, the search performed by the **FindAtom** function is not case sensitive.

If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, GlobalAddAtom, GlobalDeleteAtom, GlobalFindAtom

1.79 FindClose

The **FindClose** function closes the specified search handle. The **FindFirstFile** and **FindNextFile** functions use the search handle to locate files with names that match a given name.

```
FindClose: procedure
(
    hFindFile:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindClose@4" );
```

Parameters

hFindFile

[in/out] File search handle. This handle must have been previously opened by the **FindFirstFile** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

After the **FindClose** function is called, the handle specified by the *hFindFile* parameter cannot be used in subsequent calls to either the **FindNextFile** or **FindClose** function.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindFirstFile, FindNextFile

---

1.80 FindCloseChangeNotification

The **FindCloseChangeNotification** function stops change notification handle monitoring.

```
FindCloseChangeNotification: procedure
(
    hChangeHandle:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindCloseChangeNotification@4" );
```

Parameters

hChangeHandle

[in] Handle to a change notification handle created by the **FindFirstChangeNotification** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

After the **FindCloseChangeNotification** function is called, the handle specified by the *hChangeHandle* parameter cannot be used in subsequent calls to either the **FindNextChangeNotification** or **FindCloseChangeNotification** function.

Change notifications can also be used in the wait functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindFirstChangeNotification, FindNextChangeNotification

---

1.81 FindFirstChangeNotification

The **FindFirstChangeNotification** function creates a change notification handle and sets up initial change notification filter conditions. A wait on a notification handle succeeds when a change matching the filter conditions occurs in the specified directory or subtree.

```
FindFirstChangeNotification: procedure
(
    lpPathName:     string;
    bWatchSubtree:  boolean;
    dwNotifyFilter: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindFirstChangeNotificationA@12" );
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to watch.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

bWatchSubtree

[in] Specifies whether the function will monitor the directory or the directory tree. If this parameter is TRUE, the function monitors the directory tree rooted at the specified directory; if it is FALSE, it monitors only the specified directory.

dwNotifyFilter

[in] Specifies the filter conditions that satisfy a change notification wait. This parameter can be one or more of the following values.

Value	Meaning
FILE_NOTIFY_CHANGE_FILE_NAME	Any file name change in the watched directory or subtree causes a change notification wait operation to return. Changes include renaming, creating, or deleting a file name.
FILE_NOTIFY_CHANGE_DIR_NAME	Any directory-name change in the watched directory or subtree causes a change notification wait operation to return. Changes include creating or deleting a directory.

FILE_NOTIFY_CHANGE_ATTRIBUTES	Any attribute change in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SIZE	Any file-size change in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_WRITE	Any change to the last write-time of files in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_SECURITY	Any security-descriptor change in the watched directory or subtree causes a change notification wait operation to return.

Return Values

If the function succeeds, the return value is a handle to a find change notification object.

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

Remarks

The wait functions can monitor the specified directory or subtree by using the handle returned by the **FindFirstChangeNotification** function. A wait is satisfied when one of the filter conditions occurs in the monitored directory or subtree.

After the wait has been satisfied, the application can respond to this condition and continue monitoring the directory by calling the **FindNextChangeNotification** function and the appropriate wait function. When the handle is no longer needed, it can be closed by using the **FindCloseChangeNotification** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindCloseChangeNotification, FindNextChangeNotification

---

1.82 FindFirstFile

The **FindFirstFile** function searches a directory for a file whose name matches the specified file name. **FindFirstFile** examines subdirectory names as well as file names.

To specify additional attributes to be used in the search, use the **FindFirstFileEx** function.

```
FindFirstFile: procedure
(
        lpFileName:     string;
    var lpFindFileData: WIN32_FIND_DATA
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__FindFirstFileA@8" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?).

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpFindFileData

[out] Pointer to the **WIN32_FIND_DATA** structure that receives information about the found file or subdirectory.

Return Values

If the function succeeds, the return value is a search handle used in a subsequent call to **FindNextFile** or **FindClose**.

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

Remarks

The **FindFirstFile** function opens a search handle and returns information about the first file whose name matches the specified pattern. After the search handle has been established, use the **FindNextFile** function to search for other files that match the same pattern. When the search handle is no longer needed, close it by using the **FindClose** function.

This function searches for files by name only; it cannot be used for attribute-based searches.

You cannot use root directories as the *lpFileName* input string for **FindFirstFile**, with or without a trailing backslash. To examine files in a root directory, use something like "C:*" and step through the directory with **FindNextFile**. To get the attributes of a root directory, use **GetFileAttributes**. Prepending the string "\\?\" does not allow access to the root directory.

Similarly, on network shares, you can use an *lpFileName* of the form "\\server\service*" but you cannot use an *lpFileName* that points to the share itself, such as "\\server\service".

To examine any directory other than a root directory, use an appropriate path to that directory, with no trailing backslash. For example, an argument of "C:\windows" will return information about the directory "C:\windows", not about any directory or file in "C:\windows". An attempt to open a search with a trailing backslash will always fail.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

The following code shows a minimal use of **FindFirstFile**.

```
#define _WIN32_WINNT 0x0400

#include "windows.h"

int
main(int argc, char *argv[])
{
  WIN32_FIND_DATA FindFileData;
  HANDLE hFind;

  printf ("Target file is %s.\n", argv[1]);
```

```
  hFind = FindFirstFile(argv[ 1], &FindFileData);

  if (hFind == INVALID_HANDLE_VALUE) {
    printf ("Invalid File Handle. Get Last Error reports %d\n", GetLastError ());
  } else {
    printf ("The first file found is %s\n", FindFileData.cFileName);
    FindClose(hFind);
  }

  return (0);
}
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindClose, FindFirstFileEx, FindNextFile, GetFileAttributes, SetFileAttributes, WIN32_FIND_DATA

1.83 FindFirstFileEx

The **FindFirstFileEx** function searches a directory for a file whose name and attributes match those specified in the function call.

```
FindFirstFileEx: procedure
(
        lpFileName:         string;
        fInfoLevelId:       FINDEX_INFO_LEVELS;
    var lpFindFileData:     var;
        fSearchOp:          FINDEX_SEARCH_OPS;
    var lpSearchFilter:     var;
        dwAdditionalFlags:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindFirstFileExA@24" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?).

In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

fInfoLevelId

[in] Specifies a **FINDEX_INFO_LEVELS** enumeration type that gives the information level of the returned data.

lpFindFileData

[out] Pointer to the buffer that receives the file data. The pointer type is determined by the level of information specified in the *fInfoLevelId* parameter.

fSearchOp

[in] Specifies a **FINDEX_SEARCH_OPS** enumeration type that gives the type of filtering to perform beyond wildcard matching.

lpSearchFilter

[in] If the specified *fSearchOp* needs structured search information, *lpSearchFilter* points to the search criteria. At this time, none of the supported *fSearchOp* values require extended search information. Therefore, this pointer must be NULL.

dwAdditionalFlags

[in] Specifies additional options for controlling the search. You can use FIND_FIRST_EX_CASE_SENSITIVE for case-sensitive searches. The default search is case insensitive. At this time, no other flags are defined.

Return Values

If the function succeeds, the return value is a search handle that can be used in a subsequent call to the **FindNextFile** or **FindClose** functions.

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

Remarks

The **FindFirstFileEx** function is provided to open a search handle and return information about the first file whose name matches the specified pattern and attributes.

If the underlying file system does not support the specified type of filtering, other than directory filtering, **FindFirstFileEx** fails with the error ERROR_NOT_SUPPORTED. The application has to use **FINDEX_SEARCH_OPS** type FileExSearchNameMatch and perform its own filtering.

After the search handle has been established, use it in the **FindNextFile** function to search for other files that match the same pattern with the same filtering being performed. When the search handle is no longer needed, it should be closed using the **FindClose** function.

You cannot use root directories as the *lpFileName* input string for **FindFirstFileEx**, with or without a trailing backslash. To examine files in a root directory, use something like "C:*" and step through the directory with **FindNextFile**. To get the attributes of a root directory, use **GetFileAttributes**. Prepending the string "\\?\" does not allow access to the root directory.

Similarly, on network shares, you can use an *lpFileName* of the form "\\server\service*" but you cannot use an *lpFileName* that points to the share itself, such as "\\server\service".

To examine any directory other than a root directory, use an appropriate path to that directory, with no trailing backslash. For example, an argument of "C:\windows" will return information about the directory "C:\windows", not about any directory or file in "C:\windows". An attempt to open a search with a trailing backslash will always fail.

The call

```
FindFirstFileEx( lpFileName,
                 FindExInfoStandard,
                 lpFindData,
                 FindExSearchNameMatch,
                 NULL,
                 0 );
```

is equivalent to the call

```
FindFirstFile( lpFileName, lpFindData);
```

The following code shows a minimal use of **FindFirstFileEx**. This program is the equivalent of the example shown in **FindFirstFile**.

```
#define _WIN32_WINNT 0x0400

#include "windows.h"

int
main(int argc, char *argv[])
{
  WIN32_FIND_DATA FindFileData;
  HANDLE hFind;

  printf ("Target file is %s.\n", argv[1]);

  hFind = FindFirstFileEx(argv[1], FindExInfoStandard, &FindFileData,
                FindExSearchNameMatch, NULL, 0 );

  if (hFind == INVALID_HANDLE_VALUE) {
    printf ("Invalid File Handle. Get Last Error reports %d\n", GetLastError ());
  } else {
    printf ("The first file found is %s\n", FindFileData.cFileName);
    FindClose(hFind);
  }

  return (0);
}
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FINDEX_INFO_LEVELS, FINDEX_SEARCH_OPS, FindFirstFile, FindNextFile, FindClose, GetFileAttributes

---

1.84 FindNextChangeNotification

The **FindNextChangeNotification** function requests that the operating system signal a change notification handle the next time it detects an appropriate change.

```
    FindNextChangeNotification: procedure
    (
        hChangeHandle:dword
    );
        stdcall;
        returns( "eax" );
```

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

Parameters

hChangeHandle

[in] Handle to a change notification handle created by the **FindFirstChangeNotification** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

After the **FindNextChangeNotification** function returns successfully, the application can wait for notification that a change has occurred by using the wait functions.

If a change occurs after a call to **FindFirstChangeNotification** but before a call to **FindNextChangeNotification**, the operating system records the change. When **FindNextChangeNotification** is executed, the recorded change immediately satisfies a wait for the change notification.

FindNextChangeNotification should not be used more than once on the same handle without using one of the wait functions. An application may miss a change notification if it uses **FindNextChangeNotification** when there is a change request outstanding.

When *hChangeHandle* is no longer needed, close it by using the **FindCloseChangeNotification** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindCloseChangeNotification, FindFirstChangeNotification

---

1.85 FindNextFile

The **FindNextFile** function continues a file search from a previous call to the **FindFirstFile** function.

```
FindNextFile: procedure
(
        hFindFile:      dword;
    var lpFindFileData: WIN32_FIND_DATA
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindNextFileA@8" );
```

Parameters

hFindFile

[in] Search handle returned by a previous call to the **FindFirstFile** function.

lpFindFileData

[out] Pointer to the **WIN32_FIND_DATA** structure that receives information about the found file or subdirectory.

The structure can be used in subsequent calls to **FindNextFile** to refer to the found file or directory.

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 no matching files can be found, the **GetLastError** function returns ERROR_NO_MORE_FILES.

Remarks

The **FindNextFile** function searches for files by name only; it cannot be used for attribute-based searches.

The order in which this function returns the file names is dependent on the file system type. With NTFS and CDFS file systems, the names are returned in alphabetical order. With FAT file systems, the names are returned in the order the files were written to the disk, which may or may not be in alphabetical order.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindClose, FindFirstFile, GetFileAttributes, SetFileAttributes, WIN32_FIND_DATA

1.86 FindResource

The **FindResource** function determines the location of a resource with the specified type and name in the specified module.

To specify a language, use the **FindResourceEx** function.

```
FindResource: procedure
(
    hModule:    dword;
    lpName:     string;
    lpType:     string
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindResourceA@12" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resource.

A value of NULL specifies the module handle associated with the image file that the operating system used to create the current process.

lpName

[in] Specifies the name of the resource. For more information, see the Remarks section.

lpType

[in] Specifies the resource type. For more information, see the Remarks section. For standard resource types, this

parameter can be one of the following values.

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource
RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry
RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource

Return Values

If the function succeeds, the return value is a handle to the specified resource's information block. To obtain a handle to the resource, pass this handle to the **`LoadResource`** function.

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

Remarks

If **`IS_INTRESOURCE`**(x) is TRUE for x = *lpName* or *lpType*, x specifies the integer identifier of the name or type of the given resource. Otherwise, those parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer identifier of the resource's name or type. For example, the string "#258" represents the integer identifier 258.

To reduce the amount of memory required for a resource, an application should refer to it by integer identifier instead of by name.

An application can use **FindResource** to find any type of resource, but this function should be used only if the application must access the binary resource data when making subsequent calls to **`LockResource`**.

To use a resource immediately, an application should use one of the following resource-specific functions to find and load the resources in one call.

Function	Action

FormatMessage	Loads and formats a message-table entry.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.
LoadMenu	Loads a menu resource.
LoadString	Loads a string-table entry.

For example, an application can use the **LoadIcon** function to load an icon for display on the screen. However, the application should use **FindResource** and **LoadResource** if it is loading the icon to copy its data to another application.

String resources are stored in sections of up to 16 strings per section. The strings in each section are stored as a sequence of counted (not null-terminated) Unicode strings. The **LoadString** function will extract the string resource from its corresponding section.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, FindResourceEx, FormatMessage, IS_INTRESOURCE, LoadAccelerators, LoadBitmap, LoadCursor, LoadIcon, LoadMenu, LoadResource, LoadString, LockResource, SizeofResource

---

1.87 FindResourceEx

The **FindResourceEx** function determines the location of the resource with the specified type, name, and language in the specified module.

```
FindResourceEx: procedure
(
    hModule:    dword;
    lpType:     string;
    lpName:     string;
    wLanguage:  word
);
    stdcall;
    returns( "eax" );
    external( "__imp__FindResourceExA@16" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resource. If this parameter is NULL, the function searches the module used to create the current process.

lpType

[in] Pointer to a null-terminated string specifying the type name of the resource. For more information, see the

Remarks section. For standard resource types, this parameter can be one of the following values.

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource
RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry
RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource

lpName

[in] Pointer to a null-terminated string specifying the name of the resource. For more information, see the Remarks section.

wLanguage

[in] Specifies the language of the resource. If this parameter is MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL), the current language associated with the calling thread is used.

To specify a language other than the current language, use the **MAKELANGID** macro to create this parameter. For more information, see **`MAKELANGID`**.

Return Values

If the function succeeds, the return value is a handle to the specified resource's information block. To obtain a handle to the resource, pass this handle to the **`LoadResource`** function.

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

Remarks

If **`IS_INTRESOURCE`**(x) is TRUE for x = *lpType* or *lpName*, x specifies the integer identifier of the type or name of the given resource. Otherwise, those parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer identifier of the resource's name or type. For example, the string "#258" represents the integer identifier 258.

To reduce the amount of memory required for a resource, an application should refer to it by integer identifier instead of by name.

An application can use **FindResourceEx** to find any type of resource, but this function should be used only if the application must access the binary resource data when making subsequent calls to the **LockResource** function.

To use a resource immediately, an application should use the following resource-specific functions to find and load the resources in one call.

Function	Action
FormatMessage	Loads and formats a message-table entry.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.
LoadMenu	Loads a menu resource.
LoadString	Loads a string-table entry.

For example, an application can use the **LoadIcon** function to load an icon for display on the screen. However, the application should use **FindResourceEx** and **LoadResource** if it is loading the icon to copy its data to another application.

String resources are stored in sections of up to 16 strings per section. The strings in each section are stored as a sequence of counted (not null-terminated) Unicode strings. The **LoadString** function will extract the string resource from its corresponding section.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, FormatMessage, IS_INTRESOURCE, LoadAccelerators, LoadBitmap, LoadCursor, LoadIcon, LoadMenu, LoadString, LoadResource, MAKELANGID

1.88 FlushConsoleInputBuffer

The **FlushConsoleInputBuffer** function flushes the console input buffer. All input records currently in the input buffer are discarded.

```
FlushConsoleInputBuffer: procedure
(
    hConsoleInput:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FlushConsoleInputBuffer@4" );
```

Parameters

hConsoleInput

[in] Handle to the console input buffer. The handle must have GENERIC_WRITE access.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetNumberOfConsoleInputEvents, PeekConsoleInput, ReadConsoleInput, WriteConsoleInput

---

1.89 FlushFileBuffers

The **FlushFileBuffers** function clears the buffers for the specified file and causes all buffered data to be written to the file.

```
FlushFileBuffers: procedure
(
    hFile:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FlushFileBuffers@4" );
```

Parameters

hFile

[in] Handle to an open file. The function flushes this file's buffers. The file handle must have GENERIC_WRITE access to the file.

If *hFile* is a handle to a communications device, the function only flushes the transmit buffer.

If *hFile* is a handle to the server end of a named pipe, the function does not return until the client has read all buffered data from the pipe.

Windows NT/2000: The function fails if *hFile* is a handle to console output. That is because console output is not buffered. The function returns FALSE, and **GetLastError** returns ERROR_INVALID_HANDLE.

Windows 95: The function does nothing if *hFile* is a handle to console output. That is because console output is not buffered. The function returns TRUE, but it does nothing.

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 **WriteFile** and **WriteFileEx** functions typically write data to an internal buffer that the operating system writes to disk on a regular basis. The **FlushFileBuffers** function writes all of the buffered information for the specified file to disk.

You can pass the same file handle used with the **_lread**, **_lwrite**, **_lcreat**, and related functions to **FlushFileBuffers**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, _lread, _lwrite, _lcreat, WriteFile, WriteFileEx

---

1.90 FlushInstructionCache

The **FlushInstructionCache** function flushes the instruction cache for the specified process.

```
FlushInstructionCache: procedure
(
        hProcess:       dword;
    var lpBaseAddress:  var;
        dwSize:         dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FlushInstructionCache@12" );
```

Parameters

hProcess

[in] Handle to a process whose instruction cache is to be flushed.

lpBaseAddress

[in] Pointer to the base of the region to be flushed. This parameter can be NULL.

dwSize

[in] Specifies the length, in bytes, of the region to be flushed if the *lpBaseAddress* parameter is not NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Windows NT/ 2000: Applications should call **FlushInstructionCache** if they generate or modify code in memory. The CPU cannot detect the change, and may execute the old code it cached.

Windows 95/98: The **FlushInstructionCache** function has no effect; it always returns TRUE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions

1.91 FlushViewOfFile

The **FlushViewOfFile** function writes to the disk a byte range within a mapped view of a file.

```
FlushViewOfFile: procedure
(
    var lpBaseAddress:          var;
        dwNumberOfBytesToFlush: SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__FlushViewOfFile@8" );
```

Parameters

lpBaseAddress

[in] Pointer to the base address of the byte range to be flushed to the disk representation of the mapped file.

dwNumberOfBytesToFlush

[in] Specifies the number of bytes to flush. If *dwNumberOfBytesToFlush* is zero, the file is flushed from the base address to the end of the mapping.

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

Flushing a range of a mapped view causes any dirty pages within that range to be written to the disk. Dirty pages are those whose contents have changed since the file view was mapped.

When flushing a memory-mapped file over a network, **FlushViewOfFile** guarantees that the data has been written from the local computer, but not that the data resides on the remote computer. The server can cache the data on the remote side. Therefore, **FlushViewOfFile** can return before the data has been physically written to disk. However, you can cause **FlushViewOfFile** to return only when the physical write is complete by specifying the FILE_FLAG_WRITE_THROUGH flag when you open the file with the **CreateFile** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, CreateFile, MapViewOfFile, UnmapViewOfFile

1.92 FoldString

The **FoldString** function maps one string to another, performing a specified transformation option.

```
FoldString: procedure
(
        dwMapFlags: dword;
        lpSrcStr:   string;
        cchSrc:     int32;
    var lpDestStr:  var;
        cchDest:    int32
);
    stdcall;
    returns( "eax" );
    external( "__imp__FoldStringA@20" );
```

Parameters

dwMapFlags

[in] A set of bit flags that indicate the type of transformation to be used during mapping. This value can be a combination of the following values.

Value	Meaning
MAP_FOLDCZONE	Fold compatibility zone characters into standard Unicode equivalents. For information about compatibility zone characters, see the following Remarks section.
MAP_FOLDDIGITS	Map all digits to Unicode characters 0 through 9.
MAP_PRECOMPOSED	Map accented characters to precomposed characters, in which the accent and base character are combined into a single character value. This value cannot be combined with MAP_COMPOSITE.
MAP_COMPOSITE	Map accented characters to composite characters, in which the accent and base character are represented by two character values. This value cannot be combined with MAP_PRECOMPOSED.
MAP_EXPAND_LIGATURES	Expand all ligature characters so that they are represented by their two-character equivalent. For example, the ligature 'æ' expands to the two characters 'a' and 'e'. This value cannot be combined with MAP_PRECOMPOSED or MAP_COMPOSITE.

lpSrcStr

[in] Pointer to the string to be mapped.

cchSrc

[in] Specifies the size, in **TCHARs**, of the *lpSrcStr* buffer. This refers to bytes for ANSI versions of the function or characters for Unicode versions. If *cchSrc* is –1, *lpSrcStr* is assumed to be null-terminated, and the length is calculated automatically.

lpDestStr

[out] Pointer to the buffer to store the mapped string.

cchDest

[in] Specifies the size, in **TCHARs**, of the *lpDestStr* buffer. If *cchDest* is zero, the function returns the number of characters required to hold the mapped string, and the buffer pointed to by *lpDestStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the destination buffer, or if the *cchDest* parameter is zero, the number of characters required to hold the mapped string. This refers to bytes for ANSI versions

of the function or characters for Unicode versions.

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

Remarks

The mapped string is null-terminated if the source string is null-terminated.

The *lpSrcStr* and *lpDestStr* pointers must not be the same. If they are the same, the function fails and **GetLastError** returns ERROR_INVALID_PARAMETER.

The compatibility zone in Unicode consists of characters in the range 0xF900 through 0xFFEF that are assigned to characters from other character-encoding standards but are actually variants of characters that are already in Unicode. The compatibility zone is used to support round-trip mapping to these standards. Applications can use the MAP_FOLDCZONE flag to avoid supporting the duplication of characters in the compatibility zone.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, LCMapString, CompareString

---

1.93 FormatMessage

The **FormatMessage** function formats a message string. The function requires a message definition as input. The message definition can come from a buffer passed into the function. It can come from a message table resource in an already-loaded module. Or the caller can ask the function to search the system's message table resource(s) for the message definition. The function finds the message definition in a message table resource based on a message identifier and a language identifier. The function copies the formatted message text to an output buffer, processing any embedded insert sequences if requested.

```
FormatMessage: procedure
(
        dwFlags:        dword;
    var lpSource:       var;
        dwMessageId:    dword;
        dwLanguageId:   dword;
    var lpBuffer:       var;
        nSize:          uns32;
    var Arguments:      var
);
    stdcall;
    returns( "eax" );
    external( "__imp__FormatMessageA@28" );
```

Parameters

dwFlags

[in] Specifies aspects of the formatting process and how to interpret the *lpSource* parameter. The low-order byte of *dwFlags* specifies how the function handles line breaks in the output buffer. The low-order byte can also spec-

ify the maximum width of a formatted output line.

You can specify a combination of the following values.

Value	Meaning
FORMAT_MESSAGE_ALLOCATE_BUFFER	Specifies that the *lpBuffer* parameter is a pointer to a **PVOID** pointer, and that the *nSize* parameter specifies the minimum number of **TCHARs** to allocate for an output message buffer. The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by *lpBuffer*. The caller should use the **LocalFree** function to free the buffer when it is no longer needed.
FORMAT_MESSAGE_IGNORE_INSERTS	Specifies that insert sequences in the message definition are to be ignored and passed through to the output buffer unchanged. This flag is useful for fetching a message for later formatting. If this flag is set, the *Arguments* parameter is ignored.
FORMAT_MESSAGE_FROM_STRING	Specifies that *lpSource* is a pointer to a null-terminated message definition. The message definition may contain insert sequences, just as the message text in a message table resource may. Cannot be used with FORMAT_MESSAGE_FROM_HMODULE or FORMAT_MESSAGE_FROM_SYSTEM.
FORMAT_MESSAGE_FROM_HMODULE	Specifies that *lpSource* is a module handle containing the message-table resource(s) to search. If this *lpSource* handle is NULL, the current process's application image file will be searched. Cannot be used with FORMAT_MESSAGE_FROM_STRING.
FORMAT_MESSAGE_FROM_SYSTEM	Specifies that the function should search the system message-table resource(s) for the requested message. If this flag is specified with FORMAT_MESSAGE_FROM_HMODULE, the function searches the system message table if the message is not found in the module specified by *lpSource*. Cannot be used with FORMAT_MESSAGE_FROM_STRING. If this flag is specified, an application can pass the result of the **GetLastError** function to retrieve the message text for a system-defined error.
FORMAT_MESSAGE_ARGUMENT_ARRAY	Specifies that the *Arguments* parameter is *not* a **va_list** structure, but instead is just a pointer to an array of values that represent the arguments.

The low-order byte of *dwFlags* can specify the maximum width of a formatted output line. Use the FORMAT_MESSAGE_MAX_WIDTH_MASK constant and bitwise Boolean operations to set and retrieve this maximum width value.

The following table shows how **FormatMessage** interprets the value of the low-order byte.

Value	Meaning
0	There are no output line width restrictions. The function stores line breaks that are in the message definition text into the output buffer.

A nonzero value other than FORMAT_MESSAGE_MAX_WIDTH_MASK	The nonzero value is the maximum number of characters in an output line. The function ignores regular line breaks in the message definition text. The function never splits a string delimited by white space across a line break. The function stores hard-coded line breaks in the message definition text into the output buffer. Hard-coded line breaks are coded with the %n escape sequence.
FORMAT_MESSAGE_MAX_WIDTH_MASK	The function ignores regular line breaks in the message definition text. The function stores hard-coded line breaks in the message definition text into the output buffer. The function generates no new line breaks.

lpSource

[in] Specifies the location of the message definition. The type of this parameter depends upon the settings in the *dwFlags* parameter.

dwFlags Setting	Parameter Type
FORMAT_MESSAGE_FROM_HMODULE	This parameter is a handle to the module that contains the message table to search.
FORMAT_MESSAGE_FROM_STRING	This parameter is a pointer to a string that consists of unformatted message text. It will be scanned for inserts and formatted accordingly.

If neither of these flags is set in *dwFlags*, then *lpSource* is ignored.

dwMessageId

[in] Specifies the message identifier for the requested message. This parameter is ignored if *dwFlags* includes FORMAT_MESSAGE_FROM_STRING.

dwLanguageId

[in] Specifies the language identifier for the requested message. This parameter is ignored if *dwFlags* includes FORMAT_MESSAGE_FROM_STRING.

If you pass a specific **LANGID** in this parameter, **FormatMessage** will return a message for that **LANGID** only. If the function cannot find a message for that **LANGID**, it returns ERROR_RESOURCE_LANG_NOT_FOUND. If you pass in zero, **FormatMessage** looks for a message for **LANGIDs** in the following order:

Language neutral

Thread **LANGID**, based on the thread's locale value

User default **LANGID**, based on the user's default locale value

System default **LANGID**, based on the system default locale value

US English

If **FormatMessage** doesn't find a message for any of the preceding **LANGIDs**, it returns any language message string that is present. If that fails, it returns ERROR_RESOURCE_LANG_NOT_FOUND.

lpBuffer

[out] Pointer to a buffer for the formatted (and null-terminated) message. If *dwFlags* includes FORMAT_MESSAGE_ALLOCATE_BUFFER, the function allocates a buffer using the **LocalAlloc** function, and places the pointer to the buffer at the address specified in *lpBuffer*.

nSize

[in] If the FORMAT_MESSAGE_ALLOCATE_BUFFER flag is not set, this parameter specifies the maximum number of **TCHARs** that can be stored in the output buffer. If FORMAT_MESSAGE_ALLOCATE_BUFFER is set, this parameter specifies the minimum number of **TCHARs** to allocate for an output buffer. For ANSI text, this is the number of bytes; for Unicode text, this is the number of characters.

Arguments

[in] Pointer to an array of values that are used as insert values in the formatted message. A %1 in the format string indicates the first value in the *Arguments* array; a %2 indicates the second argument; and so on.

The interpretation of each value depends on the formatting information associated with the insert in the message definition. The default is to treat each value as a pointer to a null-terminated string.

By default, the *Arguments* parameter is of type **va_list***, which is a language- and implementation-specific data type for describing a variable number of arguments. If you do not have a pointer of type **va_list***, then specify the FORMAT_MESSAGE_ARGUMENT_ARRAY flag and pass a pointer to an array of values; those values are input to the message formatted as the insert values. Each insert must have a corresponding element in the array.

Windows 95: No single insertion string may exceed 1023 characters in length.

Return Values

If the function succeeds, the return value is the number of **TCHARs** stored in the output buffer, excluding the terminating null character.

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

Remarks

The **FormatMessage** function can be used to obtain error message strings for the system error codes returned by **GetLastError**, as shown in the following sample code.

```
LPVOID lpMsgBuf;
FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS,
    NULL,
    GetLastError(),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
    (LPTSTR) &lpMsgBuf,
    0,
    NULL
);
// Process any inserts in lpMsgBuf.
// ...
// Display the string.
MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
// Free the buffer.
LocalFree( lpMsgBuf );
```

Within the message text, several escape sequences are supported for dynamically formatting the message. These escape sequences and their meanings are shown in the following table. All escape sequences start with the percent character (%).

Escape Sequence	Meaning
%0	Terminates a message text line without a trailing newline character. This escape sequence can be used to build up long lines or to terminate the message itself without a trailing newline character. It is useful for prompt messages.

%*n***!***printf format string***!**	Identifies an insert. The value of *n* can be in the range 1 through 99. The **printf** format string (which must be bracketed by exclamation marks) is optional and defaults to **!s!** if not specified. The **printf** format string can contain the * specifier for either the precision or the width component. If * is specified for one component, the **FormatMessage** function uses insert %*n*+1; it uses %*n*+2 if * is specified for both components. Floating-point **printf** format specifiers — e, E, f, and g — are not supported. The workaround is to to use the **sprintf** function to format the floating-point number into a temporary buffer, then use that buffer as the insert string.

Any other nondigit character following a percent character is formatted in the output message without the percent character. Following are some examples:

Format string	Resulting output
%%	A single percent sign in the formatted message text.
%n	A hard line break when the format string occurs at the end of a line. This format string is useful when **FormatMessage** is supplying regular line breaks so the message fits in a certain width.
%*space*	A space in the formatted message text. This format string can be used to ensure the appropriate number of trailing spaces in a message text line.
%.	A single period in the formatted message text. This format string can be used to include a single period at the beginning of a line without terminating the message text definition.
%!	A single exclamation point in the formatted message text. This format string can be used to include an exclamation point immediately after an insert without its being mistaken for the beginning of a **printf** format string.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions, MESSAGETABLE Resource, Message Compiler

---

1.94 FreeConsole

The **FreeConsole** function detaches the calling process from its console.

```
FreeConsole: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__FreeConsole@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If other processes share the console, the console is not destroyed, but the calling process cannot refer to it.

A process can use **FreeConsole** to detach itself from its current console, and then it can call the **AllocConsole** function to create a new console.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Wincon.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, AllocConsole

---

1.95 FreeEnvironmentStrings

The **FreeEnvironmentStrings** function frees a block of environment strings.

```
FreeEnvironmentStrings: procedure
(
    lpszEnvironmentBlock:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__FreeEnvironmentStringsA@4" );
```

Parameters

lpszEnvironmentBlock

[in] Pointer to a block of environment strings. The pointer to the block must be obtained by calling the **GetEnvironmentStrings** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

When **GetEnvironmentStrings** is called, it allocates memory for a block of environment strings. When the block is no longer needed, it should be freed by calling **FreeEnvironmentStrings**.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Processes and Threads Overview, Process and Thread Functions, GetEnvironmentStrings

---

1.96 FreeLibrary

The **FreeLibrary** function decrements the reference count of the loaded dynamic-link library (DLL). When the reference count reaches zero, the module is unmapped from the address space of the calling process and the handle is no longer valid.

```
FreeLibrary: procedure
(
    hModule:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FreeLibrary@4" );
```

Parameters

hModule

[in, out] Handle to the loaded DLL module. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

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

Each process maintains a reference count for each loaded library module. This reference count is incremented each time **LoadLibrary** is called and is decremented each time **FreeLibrary** is called. A DLL module loaded at process initialization due to load-time dynamic linking has a reference count of one. This count is incremented if the same module is loaded by a call to **LoadLibrary**.

Before unmapping a library module, the system enables the DLL to detach from the process by calling the DLL's **DllMain** function, if it has one, with the DLL_PROCESS_DETACH value. Doing so gives the DLL an opportunity to clean up resources allocated on behalf of the current process. After the entry-point function returns, the library module is removed from the address space of the current process.

It is not safe to call **FreeLibrary** from **DllMain**. For more information, see the Remarks section in **DllMain**.

Calling **FreeLibrary** does not affect other processes using the same library module.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, DllMain, FreeLibraryAndExitThread, GetModuleHandle, LoadLibrary

1.97 FreeLibraryAndExitThread

The **FreeLibraryAndExitThread** function decrements the reference count of a loaded dynamic-link library (DLL) by one, and then calls **ExitThread** to terminate the calling thread. The function does not return.

The **FreeLibraryAndExitThread** function gives threads that are created and executed within a dynamic-link library an opportunity to safely unload the DLL and terminate themselves.

```
FreeLibraryAndExitThread: procedure
(
    hModule:    dword;
    dwExitCode: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__FreeLibraryAndExitThread@8" );
```

Parameters

hModule

[in] Handle to the DLL module whose reference count the function decrements. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

dwExitCode

[in] Specifies the exit code for the calling thread.

Return Values

The function has no return value. The function does not return. Invalid *hModule* handles are ignored.

Remarks

The **FreeLibraryAndExitThread** function is implemented as:

```
FreeLibrary(hModule);
ExitThread(dwExitCode);
```

Refer to the reference pages for **FreeLibrary** and **ExitThread** for further information on those functions.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, FreeLibrary, ExitThread, DisableThreadLibraryCalls

1.98 FreeResource

The **FreeResource** function is obsolete. It is provided only for compatibility with 16-bit Windows. It is not necessary for Win32-based applications to free resources loaded by using the **LoadResource** function. A resource is automatically freed when its module is unloaded.

To save memory and decrease the size of your process's working set, Win32-based applications should release the memory associated with resources by calling the following functions.

Resource	Release function
Accelerator table	DestroyAcceleratorTable
Bitmap	DeleteObject
Cursor	DestroyCursor
Icon	DestroyIcon
Menu	DestroyMenu

See Also
Resources Overview, Resource Functions

1.99 GenerateConsoleCtrlEvent

The **GenerateConsoleCtrlEvent** function sends a specified signal to a console process group that shares the console associated with the calling process.

```
GenerateConsoleCtrlEvent: procedure
(
    dwCtrlEvent:        dword;
    dwProcessGroupId:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GenerateConsoleCtrlEvent@8" );
```

Parameters

dwCtrlEvent

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

Value	Meaning
CTRL_C_EVENT	Generates a CTRL+C signal.
CTRL_BREAK_EVENT	Generates a CTRL+BREAK signal.

dwProcessGroupId

[in] Specifies the identifier of the process group that receives the signal. A process group is created when the CREATE_NEW_PROCESS_GROUP flag is specified in a call to the **CreateProcess** function. The process identifier of the new process is also the process group identifier of a new process group. The process group includes all processes that are descendants of the root process. Only those processes in the group that share the same console as the calling process receive the signal. In other words, if a process in the group creates a new console, that process does not receive the signal, nor do its descendants.

If this parameter is zero, the signal is generated in all processes that share the console of the calling process.

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

GenerateConsoleCtrlEvent causes the control handler functions of processes in the target group to be called. All console processes have a default handler function that calls the **ExitProcess** function. A console process can use the **SetConsoleCtrlHandler** function to install or remove other handler functions.

SetConsoleCtrlHandler can also enable an inheritable attribute that causes the calling process to ignore CTRL+C signals. If **GenerateConsoleCtrlEvent** sends a CTRL+C signal to a process for which this attribute is enabled, the handler functions for that process are not called. CTRL+BREAK signals always cause the handler functions to be called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CreateProcess, ExitProcess, SetConsoleCtrl-Handler

---

1.100 GetACP

The **GetACP** function retrieves the current ANSI code-page identifier for the system.

```
GetACP: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetACP@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the current ANSI code-page identifier for the system, or a default identifier if no code page is current.

Remarks

The following are the ANSI code-page identifiers.

Identifier	Meaning
874	Thai
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1200	Unicode (BMP of ISO 10646)
1250	Windows 3.1 Eastern European

1251	Windows 3.1 Cyrillic
1252	Windows 3.1 Latin 1 (US, Western Europe)
1253	Windows 3.1 Greek
1254	Windows 3.1 Turkish
1255	Hebrew
1256	Arabic
1257	Baltic

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Windows 2000: The return value for the Indic languages is 0, because they are Unicode only.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetCPInfo, GetOEMCP

1.101 GetAtomName

The **GetAtomName** function retrieves a copy of the character string associated with the specified local atom.

```
GetAtomName: procedure
(
        nAtom:      ATOM;
    VAR lpBuffer:   var;
        nSize:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetAtomNameA@12" );
```

Parameters

nAtom

[in] Specifies the local atom that identifies the character string to be retrieved.

lpBuffer

[out] Pointer to the buffer for the character string.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer. For ANSI versions of the function this is the number of bytes, while for wide-character (Unicode) versions this is the number of characters.

Return Values

If the function succeeds, the return value is the length of the string copied to the buffer, in **TCHARs**, not including the terminating null character.

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

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GlobalAddAtom, GlobalDeleteAtom, GlobalFindAtom, GlobalGetAtomName, MAKEINTATOM

1.102 GetBinaryType

The **GetBinaryType** function determines whether a file is executable, and if so, what type of executable file it is. That last property determines which subsystem an executable file runs under.

```
GetCPInfo: procedure
(
        lpApplicationName:  string;
    var lpBinaryType:       dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCPInfo@8" );
```

Parameters

lpApplicationName

[in] Pointer to a null-terminated string that contains the full path of the file whose binary type the function shall determine.

In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

lpBinaryType

[out] Pointer to a variable to receive information about the executable type of the file specified by *lpApplicationName*. The function adjusts a set of bit flags in this variable. The following bit flag constants are defined.

Value	Description
SCS_32BIT_BINARY	A Win32-based application
SCS_DOS_BINARY	An MS-DOS – based application
SCS_OS216_BINARY	A 16-bit OS/2-based application
SCS_PIF_BINARY	A PIF file that executes an MS-DOS – based application

SCS_POSIX_BINARY	A POSIX – based application
SCS_WOW_BINARY	A 16-bit Windows-based application

Return Values

If the file is executable, the return value is nonzero. The function sets the variable pointed to by *lpBinaryType* to indicate the file's executable type.

If the function is not executable, or if the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

As an alternative, you can obtain the same information by calling the SHGetFileInfo function, passing the SHGFI_EXETYPE flag in the *uFlags* parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions

1.103 GetCPInfoEx

The **GetCPInfoEx** function retrieves information about any valid installed or available code page.

```
GetCPInfoEx: procedure
(
        CodePage:   uns32;
        dwFlags:    dword;
    var lpCPInfoEx: CPINFOEX
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCPInfoExA@12" );
```

Parameters

CodePage

[in] Specifies the code page about which information is to be retrieved. You can specify the code-page identifier for any installed or available code page, or you can specify one of the following predefined values.

Value	Meaning
CP_ACP	Use the system default–ANSI code page.
CP_MACCP	**Windows NT/2000:** Use the system default–Macintosh code page.
CP_OEMCP	Use the system default–OEM code page.

dwFlags

Reserved. Must be zero.

lpCPInfoEx

[out] Pointer to a **CPINFOEX** structure that receives information about the code page.

Return Values

If the function succeeds, the return values is a nonzero value.

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

Remarks

If the specified code page is not installed or not available, the **GetCPInfoEx** function sets the last-error value to ERROR_INVALID_PARAMETER.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetACP, GetOEMCP, CPINFOEX

1.104 GetCommConfig

The **GetCommConfig** function retrieves the current configuration of a communications device.

```
GetCommConfig: procedure
(
        hCommDev:   dword;
    var lpCC:       COMMCONFIG;
    var lpdwSize:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommConfig@12" );
```

Parameters

hCommDev

[in] Handle to the open communications device.

lpCC

[out] Pointer to a buffer that receives a **COMMCONFIG** structure.

lpdwSize

[in/out] Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by *lpCC*. When the function returns, the variable contains the number of bytes copied if the function succeeds, or the number of bytes required if the buffer was too small.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, SetCommConfig, COMMCONFIG

1.105 GetCommMask

The **GetCommMask** function retrieves the value of the event mask for a specified communications device.

```
GetCommMask: procedure
(
        hFile:      dword;
    var lpEvtMask:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommMask@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpEvtMask

[out] Pointer to the variable to be filled with a mask of events that are currently enabled. This parameter can be one or more of the following values.

Value	Meaning
EV_BREAK	A break was detected on input.
EV_CTS	The CTS (clear-to-send) signal changed state.
EV_DSR	The DSR (data-set-ready) signal changed state.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_EVENT1	An event of the first provider-specific type occurred.
EV_EVENT2	An event of the second provider-specific type occurred.
EV_PERR	A printer error occurred.
EV_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state.
EV_RX80FULL	The receive buffer is 80 percent full.
EV_RXCHAR	A character was received and placed in the input buffer.

EV_RXFLAG	The event character was received and placed in the input buffer. The event character is specified in the device's **DCB** structure, which is applied to a serial port by using the **SetCommState** function.
EV_TXEMPTY	The last character in the output buffer was sent.

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 **GetCommMask** function uses a mask variable to indicate the set of events that can be monitored for a particular communications resource. A handle to the communications resource can be specified in a call to the **WaitCommEvent** function, which waits for one of the events to occur. To modify the event mask of a communications resource, use the **SetCommMask** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, DCB, SetCommMask, WaitCommEvent

---

1.106 GetCommModemStatus

The **GetCommModemStatus** function retrieves modem control-register values.

```
GetCommModemStatus: procedure
(
        hFile:          dword;
    var lpModemStat:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommModemStatus@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpModemStat

[out] Pointer to a variable that specifies the current state of the modem control-register values. This parameter can be one or more of the following values.

Value	Meaning
MS_CTS_ON	The CTS (clear-to-send) signal is on.
MS_DSR_ON	The DSR (data-set-ready) signal is on.

MS_RING_ON	The ring indicator signal is on.
MS_RLSD_ON	The RLSD (receive-line-signal-detect) signal is on.

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 **GetCommModemStatus** function is useful when you are using the **WaitCommEvent** function to monitor the CTS, RLSD, DSR, or ring indicator signals. To detect when these signals change state, use **WaitCommEvent** and then use **GetCommModemStatus** to determine the state after a change occurs.

The function fails if the hardware does not support the control-register values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, WaitCommEvent

---

1.107 GetCommProperties

The **GetCommProperties** function retrieves information about the communications properties for a specified communications device.

```
GetCommProperties: procedure
(
        hFile:      dword;
    var lpCommProp: COMMPROP
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommProperties@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpCommProp

[out] Pointer to a **COMMPROP** structure in which the communications properties information is returned. This information can be used in subsequent calls to the **SetCommState, SetCommTimeouts**, or **SetupComm** function to configure the communications device.

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 **GetCommProperties** function returns information from a device driver about the configuration settings that are

supported by the driver.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, COMMPROP, CreateFile, SetCommState, SetCommTimeouts, SetupComm

---

1.108 GetCommState

The **GetCommState** function retrieves the current control settings for a specified communications device.

```
GetCommState: procedure
(
        hFile:  dword;
    var lpDCB:  DCB
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommState@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpDCB

[out] Pointer to a **DCB** structure that receives the control settings information.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, DCB, SetCommState

---

1.109 GetCommTimeouts

The **GetCommTimeouts** function retrieves the time-out parameters for all read and write operations on a specified communications device.

```
GetCommTimeouts: procedure
```

```
(
        hFile:          dword;
    var lpCommTimeouts: COMMTIMEOUTS
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommTimeouts@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpCommTimeouts

[out] Pointer to a **COMMTIMEOUTS** structure in which the time-out information is returned.

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

For more information about time-out values for communications devices, see the **SetCommTimeouts** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, COMMTIMEOUTS, SetCommTimeouts

---

1.110 GetCommandLine

The **GetCommandLine** function retrieves the command-line string for the current process.

```
GetCommandLine: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetCommandLineA@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a pointer to the command-line string (zero-terminated) for the current process.

Remarks

ANSI console processes written in C can use the *argc* and *argv* arguments of the **main** function to access the command-line arguments. ANSI GUI applications can use the *lpCmdLine* parameter of the **WinMain** function to access the command-line string, excluding the program name. The reason that **main** and **WinMain** cannot return Unicode strings is that *argc*, *argv*, and *lpCmdLine* use the **LPSTR** data type for parameters, not the **LPTSTR** data type. The

GetCommandLine function can be used to access Unicode strings, because it uses the **LPTSTR** data type.

To convert the command line to an *argv* style array of strings, call the **CommandLineToArgvW** function.

Note The name of the executable in the command line that the operating system provides to a process is not necessarily identical to that in the command line that the calling process gives to the **CreateProcess** function. The operating system may prepend a fully qualified path to an executable name that is provided without a fully qualified path.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CommandLineToArgvW, CreateProcess, WinMain

---

1.111 GetCompressedFileSize

The **GetCompressedFileSize** function retrieves the actual number of bytes of disk storage used to store a specified file. If the file is located on a volume that supports compression, and the file is compressed, the value obtained is the compressed size of the specified file. If the file is located on a volume that supports sparse files, and the file is a sparse file, the value obtained is the sparse size of the specified file.

```
GetCompressedFileSize: procedure
(
        lpFileName:     string;
    var lpFileSizeHigh: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCompressedFileSizeA@8" );
```

Parameters

lpFileName

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

Do not specify the name of a file on a nonseeking device, such as a pipe or a communications device, as its file size has no meaning.

lpFileSizeHigh

[out] Pointer to a variable that receives the high-order **DWORD** of the compressed file size. The function's return value is the low-order **DWORD** of the compressed file size.

This parameter can be NULL if the high-order **DWORD** of the compressed file size is not needed. Files less than 4 gigabytes in size do not need the high-order **DWORD**.

Return Values

If the function succeeds, the return value is the low-order **DWORD** of the actual number of bytes of disk storage used to store the specified file, and if *lpFileSizeHigh* is non-NULL, the function puts the high-order **DWORD** of that actual value into the **DWORD** pointed to by that parameter. This is the compressed file size for compressed files, the actual file size for noncompressed files.

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

If the return value is INVALID_FILE_SIZE and *lpFileSizeHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded (value is NO_ERROR) or failed (value is other than NO_ERROR).

Remarks

An application can determine whether a volume is compressed by calling **GetVolumeInformation**, then checking the status of the FS_VOL_IS_COMPRESSED flag in the **DWORD** value pointed to by that function's *lpFileSystemFlags* parameter.

If the file is not located on a volume that supports compression or sparse files, or if the file is not compressed or a sparse file, the value obtained is the actual file size, the same as the value returned by a call to **GetFileSize**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File Systems Overview, File System Functions, GetFileSize, GetVolumeInformation

---

1.112 GetComputerName

The **GetComputerName** function retrieves the NetBIOS name of the local computer. This name is established at system startup, when the system reads it from the registry.

If the local computer is a node in a cluster, **GetComputerName** returns the name of the node.

Windows 2000: GetComputerName retrieves only the NetBIOS name of the local computer. To retrieve the DNS host name, DNS domain name, or the fully qualified DNS name, call the **GetComputerNameEx** function.

Windows 2000: Additional information is provided by the **IADsADSystemInfo** interface.

```
GetComputerName: procedure
(
    var lpBuffer:   var;
    var lpnSize:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetComputerNameA@8" );
```

Parameters

lpBuffer

[out] Pointer to a buffer that receives a null-terminated string containing the computer name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

lpnSize

[in/out] On input, specifies the size, in **TCHARs**, of the buffer. On output, receives the number of **TCHARs** copied to the destination buffer, not including the terminating null character.

If the buffer is too small, the function fails and **GetLastError** returns ERROR_BUFFER_OVERFLOW.

Windows 95/98: GetComputerName fails if the input size is less than MAX_COMPUTERNAME_LENGTH + 1.

Return Values

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

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

Remarks

The **GetComputerName** function retrieves the NetBIOS name established at system startup. Name changes made by the **SetComputerName** or **SetComputerNameEx** functions do not take effect until the user restarts the computer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetComputerNameEx, SetComputerName, SetComputerNameEx

---

1.113 GetConsoleCP

Windows NT/2000: The **GetConsoleCP** function retrieves the input code page used by the console associated with the calling process. A console uses its input code page to translate keyboard input into the corresponding character value.

Windows 95: On Japanese and Korean implementations of Windows 95, the **GetConsoleCP** function returns the VM code page, because the OEM code page can be either 437 or DBCS. On all other implementations of Windows 95, the **GetConsoleCP** function returns the OEM code page.

```
GetConsoleCP: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetConsoleCP@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a code that identifies the code page.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

To set a console's input code page, use the **SetConsoleCP** function. To set and query a console's output code page, use the **SetConsoleOutputCP** and **GetConsoleOutputCP** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleOutputCP, SetConsoleCP, SetConsoleOutputCP

1.114 GetConsoleCursorInfo

The **GetConsoleCursorInfo** function retrieves information about the size and visibility of the cursor for the specified console screen buffer.

```
GetConsoleCursorInfo: procedure
(
        hConsoleOutput:         dword;
    var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetConsoleCursorInfo@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_READ access.

lpConsoleCursorInfo

[out] Pointer to a **CONSOLE_CURSOR_INFO** structure in which information about the console's cursor is returned.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CONSOLE_CURSOR_INFO, SetConsoleCursorInfo

---

1.115 GetConsoleMode

The **GetConsoleMode** function retrieves the current input mode of a console's input buffer or the current output mode of a console screen buffer.

```
GetConsoleMode: procedure
(
        hConsoleHandle: dword;
    var lpMode:         dword
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__GetConsoleMode@8" );
```

Parameters

hConsoleHandle

[in] Handle to a console input buffer or a screen buffer. The handle must have GENERIC_READ access.

lpMode

[out] Pointer to a variable that indicates the current mode of the specified buffer.

If the *hConsoleHandle* parameter is an input handle, the mode can be a combination of the following values. When a console is created, all input modes except ENABLE_WINDOW_INPUT are enabled by default.

Value	Meaning
ENABLE_LINE_INPUT	The **`ReadFile`** or **`ReadConsole`** function returns only when a carriage return character is read. If this mode is disabled, the functions return when one or more characters are available.
ENABLE_ECHO_INPUT	Characters read by the **ReadFile** or **ReadConsole** function are written to the active screen buffer as they are read. This mode can be used only if the ENABLE_LINE_INPUT mode is also enabled.
ENABLE_PROCESSED_INPUT	CTRL+C is processed by the system and is not placed in the input buffer. If the input buffer is being read by **`ReadFile`** or **`ReadConsole`**, other control keys are processed by the system and are not returned in the **ReadFile** or **ReadConsole** buffer. If the ENABLE_LINE_INPUT mode is also enabled, backspace, carriage return, and linefeed characters are handled by the system.
ENABLE_WINDOW_INPUT	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer by applications using the **`ReadConsoleInput`** function, but not by those using **ReadFile** or **ReadConsole**.
ENABLE_MOUSE_INPUT	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. These events are discarded by **`ReadFile`** or **`ReadConsole`**, even when this mode is enabled.

If the *hConsoleHandle* parameter is a screen buffer handle, the mode can be a combination of the following values. When a screen buffer is created, both output modes are enabled by default.

Value	Meaning
ENABLE_PROCESSED_OUTPUT	Characters written by the **`WriteFile`** or **`WriteConsole`** function or echoed by the **`ReadFile`** or **`ReadConsole`** function are parsed for ASCII control sequences, and the correct action is performed. Backspace, tab, bell, carriage return, and linefeed characters are processed.

ENABLE_WRAP_AT_EOL_OUTPUT	When writing with **WriteFile** or **WriteConsole** or echoing with **ReadFile** or **ReadConsole**, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the screen buffer to scroll up (discarding the top row of the screen buffer) when the cursor advances beyond the last row in the screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

A console consists of an input buffer and one or more screen buffers. The mode of a console buffer determines how the console behaves during input or output (I/O) operations. One set of flag constants is used with input handles, and another set is used with screen buffer (output) handles. Setting the output modes of one screen buffer does not affect the output modes of other screen buffers.

The ENABLE_LINE_INPUT and ENABLE_ECHO_INPUT modes only affect processes that use **ReadFile** or **ReadConsole** to read from the console's input buffer. Similarly, the ENABLE_PROCESSED_INPUT mode primarily affects **ReadFile** and **ReadConsole** users, except that it also determines whether CTRL+C input is reported in the input buffer (to be read by the **ReadConsoleInput** function) or is passed to a function defined by the application.

The ENABLE_WINDOW_INPUT and ENABLE_MOUSE_INPUT modes determine whether user interactions involving window resizing and mouse actions are reported in the input buffer or discarded. These events can be read by **ReadConsoleInput**, but they are always filtered by **ReadFile** and **ReadConsole**.

The ENABLE_PROCESSED_OUTPUT and ENABLE_WRAP_AT_EOL_OUTPUT modes only affect processes using **ReadFile** or **ReadConsole** and **WriteFile** or **WriteConsole**.

To change a console's I/O modes, call **SetConsoleMode** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, ReadConsole, ReadConsoleInput, ReadFile, SetConsoleMode, WriteConsole, WriteFile

---

1.116 GetConsoleOutputCP

Windows NT/2000: The **GetConsoleOutputCP** function retrieves the output code page used by the console associated with the calling process. A console uses its output code page to translate the character values written by the various output functions into the images displayed in the console window.

Windows 95: On Japanese and Korean implementations of Windows 95, the **GetConsoleOutputCP** function returns the VM code page, because the OEM code page can be either 437 or DBCS. On all other implementations of Windows 95, the **GetConsoleOutputCP** function returns the OEM code page.

```
GetConsoleOutputCP: procedure;
```

```
        stdcall;
        returns( "eax" );
        external( "__imp__GetConsoleOutputCP@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a code that identifies the code page.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

To set a console's output code page, use the **SetConsoleOutputCP** function. To set and query a console's input code page, use the **SetConsoleCP** and **GetConsoleCP** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleCP, SetConsoleCP, SetConsoleOutputCP

1.117 GetConsoleScreenBufferInfo

The **GetConsoleScreenBufferInfo** function retrieves information about the specified console screen buffer.

```
    GetConsoleScreenBufferInfo: procedure
    (
            handle: dword;
        var csbi:   CONSOLE_SCREEN_BUFFER_INFO
    );
        stdcall;
        returns( "eax" );
        external( "__imp__GetConsoleScreenBufferInfo@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_READ access.

lpConsoleScreenBufferInfo

[out] Pointer to a **CONSOLE_SCREEN_BUFFER_INFO** structure in which the screen buffer information is returned.

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 rectangle returned in the **srWindow** member of the **CONSOLE_SCREEN_BUFFER_INFO** structure can be modified and then passed to the **SetConsoleWindowInfo** function to scroll the screen buffer in the window, to change the size of the window, or both.

All coordinates returned in the **CONSOLE_SCREEN_BUFFER_INFO** structure are in character-cell coordinates, where the origin (0, 0) is at the upper-left corner of the screen buffer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CONSOLE_SCREEN_BUFFER_INFO, GetLargestConsoleWindowSize, SetConsoleCursorPosition, SetConsoleScreenBufferSize, SetConsoleWindowInfo

---

1.118 GetConsoleTitle

The **GetConsoleTitle** function retrieves the title bar string for the current console window.

```
GetConsoleTitle: procedure
(
    var lpConsoleTitle: var;
        nSize:          dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetConsoleTitleA@8" );
```

Parameters

lpConsoleTitle

[out] Pointer to a buffer that receives a null-terminated string containing the text that appears in the title bar of the console window.

nSize

[in] Specifies the size, in characters, of the buffer pointed to by the *lpConsoleTitle* parameter.

Return Values

If the function succeeds, the return value is the length, in characters, of the string copied to the buffer.

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

Remarks

To set the title bar string for a console window, use the **SetConsoleTitle** function.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, SetConsoleCP, SetConsoleOutputCP, SetConsoleTitle

---

1.119 GetConsoleWindow

The **GetConsoleWindow** function retrieves the window handle used by the console associated with the calling process.

```
GetConsoleWindow: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetConsoleWindow@0" );
```

Parameters

None.

Return Values

The return value is a handle to the window used by the console associated with the calling process or **NULL** if there is no such associated console.

See Also

Consoles and Character Support Overview, Console Functions

1.120 GetCurrencyFormat

The **GetCurrencyFormat** function formats a number string as a currency string for a specified locale.

```
GetCurrencyFormat: procedure
(
        Locale:         LCID;
        dwFlags:        dword;
        lpValue:        string;
    var lpFormat:       CURRENCYFMT;
    var lpCurrencyStr:  var;
        cchCurrency:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrencyFormatA@24" );
```

Parameters

Locale

[in] Specifies the locale for which the currency string is to be formatted. If *lpFormat* is NULL, the function formats the string according to the currency format for this locale. If *lpFormat* is not NULL, the function uses the locale only for formatting information not specified in the **CURRENCYFMT** structure (for example, the locale's string value for the negative sign).

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined val-

ues.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Controls the operation of the function. If *lpFormat* is non-NULL, this parameter must be zero.

If *lpFormat* is NULL, you can specify the LOCALE_NOUSEROVERRIDE flag to format the string using the system default currency format for the specified locale; or you can specify zero to format the string using any user overrides to the locale's default currency format.

lpValue

[in] Pointer to a null-terminated string containing the number string to format.

This string can contain only the following characters:

Characters '0' through '9'.

One decimal point (dot) if the number is a floating-point value.

A minus sign in the first character position if the number is a negative value.

All other characters are invalid. The function returns an error if the string pointed to by *lpValue* deviates from these rules.

lpFormat

[in] Pointer to a **CURRENCYFMT** structure that contains currency formatting information. All members in the structure pointed to by *lpFormat* must contain appropriate values.

If *lpFormat* is NULL, the function uses the currency format of the specified locale.

lpCurrencyStr

[out] Pointer to a buffer that receives the formatted currency string.

cchCurrency

[in] Specifies the size, in **TCHARs**, of the *lpCurrencyStr* buffer. If *cchCurrency* is zero, the function returns the number of **TCHARs** required to hold the formatted currency string, and the buffer pointed to by *lpCurrencyStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the buffer pointed to by *lpCurrencyStr*, or if the *cchCurrency* parameter is zero, the number of bytes or characters required to hold the formatted currency string. The count includes the terminating null.

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

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetNumberFormat, CURRENCYFMT

1.121 GetCurrentDirectory

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

```
GetCurrentDirectory: procedure
(
        nBufferLength:  dword;
    var lpBuffer:       var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrentDirectoryA@8" );
```

Parameters

nBufferLength

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

lpBuffer

[out] Pointer to the buffer that receives 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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateDirectory, GetSystemDirectory, GetWindowsDirectory, RemoveDirectory, SetCurrentDirectory

---

1.122 GetCurrentProcess

The **GetCurrentProcess** function retrieves a pseudo handle for the current process.

```
GetCurrentProcess: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrentProcess@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a pseudo handle to the current process.

Remarks

A pseudo handle is a special constant that is interpreted as the current process handle. The calling process can use this handle to specify its own process whenever a process handle is required. Pseudo handles are not inherited by child processes.

This handle has the maximum possible access to the process object. For systems that support security descriptors, this is the maximum access allowed by the security descriptor for the calling process. For systems that do not support security descriptors, this is PROCESS_ALL_ACCESS. For more information, see Process Security and Access Rights.

A process can create a "real" handle to itself that is valid in the context of other processes, or that can be inherited by other processes, by specifying the pseudo handle as the source handle in a call to the **DuplicateHandle** function. A process can also use the **OpenProcess** function to open a real handle to itself.

The pseudo handle need not be closed when it is no longer needed. Calling the **`CloseHandle`** function with a pseudo handle has no effect. If the pseudo handle is duplicated by **`DuplicateHandle`**, the duplicate handle must be closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CloseHandle, DuplicateHandle, GetCurrentProcessId, GetCurrentThread, OpenProcess

---

1.123 GetCurrentProcessId

The **GetCurrentProcessId** function retrieves the process identifier of the calling process.

```
GetCurrentProcessId: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrentProcessId@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the process identifier of the calling process.

Remarks

Until the process terminates, the process identifier uniquely identifies the process throughout the system.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetCurrentProcess, OpenProcess

---

1.124 GetCurrentThread

The **GetCurrentThread** function retrieves a pseudo handle for the current thread.

```
GetCurrentThread: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrentThread@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a pseudo handle for the current thread.

Remarks

A pseudo handle is a special constant that is interpreted as the current thread handle. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudo handles are not inherited by child processes.

This handle has the maximum possible access to the thread object. For systems that support security descriptors, this is the maximum access allowed by the security descriptor for the calling process. For systems that do not support security descriptors, this is THREAD_ALL_ACCESS.

The function cannot be used by one thread to create a handle that can be used by other threads to refer to the first thread. The handle is always interpreted as referring to the thread that is using it. A thread can create a "real" handle to itself that can be used by other threads, or inherited by other processes, by specifying the pseudo handle as the source handle in a call to the **DuplicateHandle** function.

The pseudo handle need not be closed when it is no longer needed. Calling the **`CloseHandle`** function with this handle has no effect. If the pseudo handle is duplicated by **`DuplicateHandle`**, the duplicate handle must be closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CloseHandle, DuplicateHandle, GetCurrentProcess, GetCurrentThreadId, OpenThread

1.125 GetCurrentThreadId

The **GetCurrentThreadId** function retrieves the thread identifier of the calling thread.

```
GetCurrentThreadId: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetCurrentThreadId@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the thread identifier of the calling thread.

Remarks

Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetCurrentThread, OpenThread

1.126 GetDateFormat

The **GetDateFormat** function formats a date as a date string for a specified locale. The function formats either a specified date or the local system date.

```
GetDateFormat: procedure
(
        Locale:     LCID;
        dwFlags:    dword;
    var lpDate:     SYSTEMTIME;
        lpFormat:   string;
    var lpDateStr:  var;
        cchDate:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetDateFormatA@24" );
```

Parameters

Locale

[in] Specifies the locale for which the date string is to be formatted. If *lpFormat* is NULL, the function formats the string according to the date format for this locale. If *lpFormat* is not NULL, the function uses the locale only for information not specified in the format picture string (for example, the locale's day and month names).

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Specifies various function options. If *lpFormat* is non-NULL, this parameter must be zero.

If *lpFormat* is NULL, you can specify a combination of the following values.

Value	Meaning
LOCALE_NOUSEROVERRIDE	If set, the function formats the string using the system default–date format for the specified locale. If not set, the function formats the string using any user overrides to the locale's default–date format.
LOCALE_USE_CP_ACP	Uses the system ANSI code page for string translation instead of the locale's code page.
DATE_SHORTDATE	Uses the short date format. This is the default. This value cannot be used with DATE_LONGDATE or DATE_YEARMONTH.
DATE_LONGDATE	Uses the long date format. This value cannot be used with DATE_SHORTDATE or DATE_YEARMONTH.
DATE_YEARMONTH	Uses the year/month format. This value cannot be used with DATE_SHORTDATE or DATE_LONGDATE.
DATE_USE_ALT_CALENDAR	Uses the alternate calendar, if one exists, to format the date string. If this flag is set, the function uses the default format for that alternate calendar, rather than using any user overrides. The user overrides will be used only in the event that there is no default format for the specified alternate calendar.
DATE_LTRREADING	Adds marks for left-to-right reading layout. This value cannot be used with DATE_RTLREADING.
DATE_RTLREADING	Adds marks for right-to-left reading layout. This value cannot be used with DATE_LTRREADING

If you do not specify either DATE_YEARMONTH, DATE_SHORTDATE, or DATE_LONGDATE, and *lpFormat* is NULL, then DATE_SHORTDATE is the default.

lpDate

[in] Pointer to a **SYSTEMTIME** structure that contains the date information to be formatted. If this pointer is NULL, the function uses the current local system date.

lpFormat

[in] Pointer to a format picture string that is used to form the date string. The format picture string must be zero terminated. If *lpFormat* is NULL, the function uses the date format of the specified locale.

Use the following elements to construct a format picture string. If you use spaces to separate the elements in the format string, these spaces will appear in the same location in the output string. The letters must be in uppercase or lowercase as shown in the table (for example, "MM" not "mm"). Characters in the format string that are enclosed in single quotation marks will appear in the same location and unchanged in the output string.

Picture	Meaning

d	Day of month as digits with no leading zero for single-digit days.
dd	Day of month as digits with leading zero for single-digit days.
ddd	Day of week as a three-letter abbreviation. The function uses the LOCALE_SABBREVDAYNAME value associated with the specified locale.
dddd	Day of week as its full name. The function uses the LOCALE_SDAYNAME value associated with the specified locale.
M	Month as digits with no leading zero for single-digit months.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as a three-letter abbreviation. The function uses the LOCALE_SABBREVMONTHNAME value associated with the specified locale.
MMMM	Month as its full name. The function uses the LOCALE_SMONTHNAME value associated with the specified locale.
y	Year as last two digits, but with no leading zero for years less than 10.
yy	Year as last two digits, but with leading zero for years less than 10.
yyyy	Year represented by full four digits.
gg	Period/era string. The function uses the CAL_SERASTRING value associated with the specified locale. This element is ignored if the date to be formatted does not have an associated era or period string.

For example, to get the date string

```
"Wed, Aug 31 94"
```

use the following picture string:

```
"ddd',' MMM dd yy"
```

lpDateStr

[out] Pointer to a buffer that receives the formatted date string.

cchDate

[in] Specifies the size, in **TCHARs**, of the *lpDateStr* buffer. If *cchDate* is zero, the function returns the number of bytes or characters required to hold the formatted date string, and the buffer pointed to by *lpDateStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the *lpDateStr* buffer, or if the *cchDate* parameter is zero, the number of bytes or characters required to hold the formatted date string. The count includes the terminating null.

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

Remarks

The earliest date supported by this function is January 1, 1601.

The day name, abbreviated day name, month name, and abbreviated month name are all localized based on the locale identifier.

The date values in the **SYSTEMTIME** structure pointed to by *lpDate* must be valid. The function checks each of the date values: year, month, day, and day of week. If the day of the week is incorrect, the function uses the correct value, and returns no error. If any of the other date values are outside the correct range, the function fails, and sets the last-error to ERROR_INVALID_PARAMETER.

The function ignores the time portions of the **SYSTEMTIME** structure pointed to by *lpDate*: **wHour**, **wMinute**, **wSecond**, and **wMilliseconds**.

If the *lpFormat* parameter is a bad format string, no errors are returned. The function simply forms the best date string that it can. For example, the only year pictures that are valid are L"yyyy" and L"yy" (the 'L' indicates a Unicode (16-bit characters) string). If L"y" is passed in, the function assumes L"yy". If L"yyy" is passed in, the function assumes L"yyyy". If more than 4 date (L"dddd") or 4 month (L"MMMM") pictures are passed in, then the function defaults to L"dddd" or L"MMMM".

Any text that should remain in its exact form in the date string should be enclosed within single quotation marks in the date format picture. The single quotation mark may also be used as an escape character to allow the single quotation mark itself to be displayed in the date string. However, the escape sequence must be enclosed within two single quotation marks. For example, to display the date as "May '93", the format string would be: L"MMMM ""yy" The first and last single quotation marks are the enclosing quotation marks. The second and third single quotation marks are the escape sequence to allow the single quotation mark to be displayed before the century.

When the date picture contains a numeric form of the day (either d or dd*)* followed by the full month name (MMMM), the genitive form of the month name is returned in the date string.

To obtain the default short and long date format without performing any actual formatting, use the **GetLocaleInfo** function with the LOCALE_SSHORTDATE or LOCALE_SLONGDATE parameter. To get the date format for an alternate calendar, use **GetLocaleInfo** with the LOCALE_IOPTIONALCALENDAR parameter. To get the date format for a particular calendar, use **GetCalendarInfo**. Also, to return all of the date formats for a particular calendar, you can use **EnumCalendarInfo** or **EnumDateFormatsEx**.

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

National Language Support Overview, National Language Support Functions, EnumCalendarInfo, EnumDateFormatsEx, GetCalendarInfo, GetLocaleInfo, GetTimeFormat, SYSTEMTIME

1.127 GetDefaultCommConfig

The **GetDefaultCommConfig** function retrieves the default configuration for the specified communications device.

```
GetDefaultCommConfig: procedure
(
        lpszName:   string;
    var lpCC:       COMMCONFIG;
    var lpdwSize:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetDefaultCommConfigA@12" );
```

Parameters

lpszName

[in] Pointer to a null-terminated string specifying the name of the device.

lpCC

[out] Pointer to a buffer that receives a **COMMCONFIG** structure.

lpdwSize

[in/out] Pointer to a variable that specifies the size, in bytes, of the buffer pointed to by *lpCC*. Upon return, the variable contains the number of bytes copied if the function succeeds, or the number of bytes required if the buffer was too small.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Communications Overview, Communication Functions, SetDefaultCommConfig, COMMCONFIG

1.128 GetDevicePowerState

The **GetDevicePowerState** function retrieves the current power state of the specified device.

```
GetDevicePowerState: procedure
(
    hDevice:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetDevicePowerState@4" );
```

Parameters

hDevice

[in] Handle to an object on the device, such as a file or socket, or a handle to the device itself.

Return Values

Otherwise, the function returns the power state in EAX.

[note: MS' documentation does not agree with the variable's declaration in kernel32.lib. Beware.]

Remarks

An application can use **GetSystemPowerState** to determine whether a disk or other device is spun up. If the device is not spun up, the application should defer accessing it.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Power Management Overview, Power Management Functions, GetSystemPowerStatus

---

1.129 GetDiskFreeSpace

The **GetDiskFreeSpace** function retrieves information about the specified disk, including the amount of free space on the disk.

This function has been superseded by the **GetDiskFreeSpaceEx** function. New Win32-based applications should use **GetDiskFreeSpaceEx**.

```
GetDiskFreeSpace: procedure
(
        lpRootPathName:         string;
    var lpSectorsPerCluster:    dword;
    var lpBytesPerSector:       dword;
    var lpNumberOfFreeClusters: dword;
    var lpTotalNumberOfClusters:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetDiskFreeSpaceA@20" );
```

Parameters

lpRootPathName

[in] Pointer 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. If this parameter is a UNC name, you must follow it with a trailing backslash. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\. However, a drive specification such as "C:" cannot have a trailing backslash.

Windows 95: The initial release of Windows 95 does not support UNC paths for the *lpszRootPathName* parameter. To query the free disk space using a UNC path, temporarily map the UNC path to a drive letter, query the free disk space on the drive, then remove the temporary mapping. **Windows 95 OSR2 and later:** UNC paths are supported.

lpSectorsPerCluster

[out] Pointer to a variable for the number of sectors per cluster.

lpBytesPerSector

[out] Pointer to a variable for the number of bytes per sector.

lpNumberOfFreeClusters

[out] Pointer to a variable for the total number of free clusters on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user disk quotas are in use, this value may be less than the total number of free clusters on the disk.

lpTotalNumberOfClusters

[out] Pointer to a variable for the total number of clusters on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user disk quotas are in use, this value may be less than the total number of clusters on the disk.

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 **GetDiskFreeSpaceEx** function lets you avoid the arithmetic required by the **GetDiskFreeSpace** function.

Windows 95:

For volumes that are larger than 2 gigabytes, the **GetDiskFreeSpace** function may return misleading values. The function caps the values stored into **lpNumberOfFreeClusters* and **lpTotalNumberOfClusters* so as to never report volume sizes that are greater than 2 gigabytes.

Even on volumes that are smaller than 2 gigabytes, the values stored into **lpSectorsPerCluster*, **lpNumberOfFreeClusters*, and **lpTotalNumberOfClusters* values may be incorrect. That is because the operating system manipulates the values so that computations with them yield the correct volume size.

Windows 95 OSR2 and Windows 98:

The **GetDiskFreeSpaceEx** function is available beginning with Windows 95 OEM Service Release 2 (OSR2), and you should use it whenever possible. The **GetDiskFreeSpaceEx** function returns correct values for all volumes, including those that are larger than 2 gigabytes.

Windows NT and Windows 2000:

GetDiskFreeSpaceEx is available on Windows NT version 4.0 and higher, including Windows 2000.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetDiskFreeSpaceEx, GetDriveType

---

1.130 GetDiskFreeSpaceEx

The **GetDiskFreeSpaceEx** function retrieves information about the amount of space available on a disk volume: the total amount of space, the total amount of free space, and the total amount of free space available to the user associated with the calling thread.

```
GetDiskFreeSpaceEx: procedure
(
        lpDirectoryName:            string;
    var lpFreeBytesAvailable:       qword;
    var lpTotalNumberOfBytes:       qword;
    var lpTotalNumberOfFreeBytes:   qword
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__GetDiskFreeSpaceExA@16" );
```

Parameters

lpDirectoryName

[in] Pointer to a null-terminated string that specifies a directory on the disk of interest. This string can be a UNC name. If this parameter is a UNC name, you must follow it with an additional backslash. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\.

If *lpDirectoryName* is NULL, the **GetDiskFreeSpaceEx** function retrieves information about the disk that contains the current directory.

Note that *lpDirectoryName* does not have to specify the root directory on a disk. The function accepts any directory on the disk.

lpFreeBytesAvailable

[out] Pointer to a variable that receives the total number of free bytes on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user quotas are in use, this value may be less than the total number of free bytes on the disk.

lpTotalNumberOfBytes

[out] Pointer to a variable that receives the total number of bytes on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user quotas are in use, this value may be less than the total number of bytes on the disk.

lpTotalNumberOfFreeBytes

[out] Pointer to a variable that receives the total number of free bytes on the disk.

This parameter can be NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Note that the values obtained by this function are of type **ULARGE_INTEGER**. Be careful not to truncate these values to 32 bits.

Windows NT and Windows 2000: GetDiskFreeSpaceEx is available on Windows NT version 4.0 and higher, including Windows 2000. See the following information for a method to determine at run time if it is available.

Windows 95 OSR2 and Windows 98: The **GetDiskFreeSpaceEx** function is available beginning with Windows 95 OEM Service Release 2 (OSR2).

To determine whether **GetDiskFreeSpaceEx** is available, call **`GetModuleHandle`** to get the handle to Kernel32.dll. Then you can call **`GetProcAddress`**.

The following code fragment shows one way to do this:

```
pGetDiskFreeSpaceEx = GetProcAddress( GetModuleHandle("kernel32.dll"),
                        "GetDiskFreeSpaceExA");

if (pGetDiskFreeSpaceEx)
{
   fResult = pGetDiskFreeSpaceEx (pszDrive,
              (PULARGE_INTEGER)&i64FreeBytesToCaller,
              (PULARGE_INTEGER)&i64TotalBytes,
              (PULARGE_INTEGER)&i64FreeBytes);
```

```
// Process GetDiskFreeSpaceEx results.
}

else
{
   fResult = GetDiskFreeSpace (pszDrive,
                &dwSectPerClust,
                &dwBytesPerSect,
                &dwFreeClusters,
                &dwTotalClusters)

// Process GetDiskFreeSpace results.

}
```

It is not necessary to call **LoadLibrary** on Kernel32.dll because it is already loaded into every Win32 process's address space.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 OSR2 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetDiskFreeSpace, GetModuleHandle, GetProcAddress

---

1.131 GetDriveType

The **GetDriveType** function determines whether a disk drive is a removable, fixed, CD-ROM, RAM disk, or network drive.

```
GetDriveType: procedure
(
    lpRootPathName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetDriveTypeA@4" );
```

Parameters

lpRootPathName

[in] Pointer to a null-terminated string that specifies the root directory of the disk to return information about. A trailing backslash is required. 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.

Value	Meaning
DRIVE_UNKNOWN	The drive type cannot be determined.
DRIVE_NO_ROOT_DIR	The root path is invalid. For example, no volume is mounted at the path.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, File I/O Functions, GetDiskFreeSpace

1.132 GetEnvironmentStrings

The **GetEnvironmentStrings** function retrieves the environment block for the current process.

```
GetEnvironmentStrings: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetEnvironmentStrings@0" );
```

Parameters

This function has no parameters.

Return Values

The return value (in EAX) is a pointer to an environment block for the current process.

Remarks

The **GetEnvironmentStrings** function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the **`GetEnvironmentVariable`** and **`SetEnvironmentVariable`** functions to retrieve or change the environment variables within this block. When the block is no longer needed, it should be freed by calling **`FreeEnvironmentStrings`**.

A process can use this function's return value to specify the environment address used by the **`CreateProcess`** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, GetEnvironmentVariable, SetEnvironmentVariable, FreeEnvironmentStrings

1.133 GetEnvironmentVariable

The **GetEnvironmentVariable** function retrieves the value of the specified variable from the environment block of the calling process. The value is in the form of a null-terminated string of characters.

```
GetEnvironmentVariable: procedure
(
        lpName:     string;
    var lpBuffer:   var;
        nSize:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetEnvironmentVariableA@12" );
```

Parameters

lpName

[in] Pointer to a null-terminated string that specifies the environment variable.

lpBuffer

[out] Pointer to a buffer to receive the value of the specified environment variable.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

If the function succeeds, the return value is the number of **TCHARs** stored into the buffer pointed to by *lpBuffer*, not including the terminating null character.

If the specified environment variable name was not found in the environment block for the current process, the return value is zero.

If the buffer pointed to by *lpBuffer* is not large enough, the return value is the buffer size, in **TCHARs**, required to hold the value string and its terminating null character.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Processes and Threads Overview, Process and Thread Functions, GetEnvironmentStrings, SetEnvironmentVariable

1.134 GetExitCodeProcess

The **GetExitCodeProcess** function retrieves the termination status of the specified process.

```
GetExitCodeProcess: procedure
(
        hProcess:   dword;
    var lpExitCode: dword
);
```

```
    stdcall;
    returns( "eax" );
    external( "__imp__GetExitCodeProcess@8" );
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have PROCESS_QUERY_INFORMATION access. For more information, see Process Security and Access Rights.

lpExitCode

[out] Pointer to a variable to receive the process termination status.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the specified process has not terminated, the termination status returned is STILL_ACTIVE. If the process has terminated, the termination status returned may be one of the following:

The exit value specified in the **ExitProcess** or **TerminateProcess** function.

The return value from the **main** or **WinMain** function of the process.

The exception value for an unhandled exception that caused the process to terminate.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, ExitProcess, ExitThread, TerminateProcess, WinMain

---

1.135 GetExitCodeThread

The **GetExitCodeThread** function retrieves the termination status of the specified thread.

```
GetExitCodeThread: procedure
(
        hThread:    dword;
    var lpExitCode: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetExitCodeThread@8" );
```

Parameters

hThread

[in] Handle to the thread.

Windows NT/2000: The handle must have THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

lpExitCode

[out] Pointer to a variable to receive the thread termination status.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the specified thread has not terminated, the termination status returned is STILL_ACTIVE. If the thread has terminated, the termination status returned may be one of the following:

The exit value specified in the **ExitThread** or **TerminateThread** function.

The return value from the thread function.

The exit value of the thread's process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, ExitThread, GetExitCodeProcess, OpenThread, TerminateThread

---

1.136 GetFileAttributes

The **GetFileAttributes** function retrieves attributes for a specified file or directory.

This function retrieves a set of FAT-style attribute information. The **GetFileAttributesEx** function can obtain other sets of file or directory attribute information.

```
GetFileAttributes: procedure
(
    lpFileName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetFileAttributesA@4" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of a file or directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

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 -1. To get extended error information, call **GetLastError**.

The attributes can be one or more of the following values.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file or directory. 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_DEVICE	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data streams in the file are encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
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_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
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_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.
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.

Remarks

When **GetFileAttributes** is called on a directory containing a volume mount point, the file attributes returned are

those of the directory where the volume mount point is set, not those of the root directory in the target mounted volume. To obtain the file attributes of the mounted volume, call **GetVolumeNameForVolumeMountPoint** to obtain the name of the target volume. Then use the resulting name in a call to **GetFileAttributes**. The results will be the attributes of the root directory on the target volume.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, File I/O Functions, DeviceIoControl, FindFirstFile, FindNextFile, GetFileAttributesEx, SetFileAttributes

---

1.137 GetFileAttributesEx

The **GetFileAttributesEx** function retrieves attributes for a specified file or directory.

```
GetFileAttributesEx: procedure
(
        lpFileName:         string;
        fInfoLevelId:       GET_FILEEX_INFO_LEVELS;
    var lpFileInformation:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetFileAttributesExA@12" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies a file or directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 98: This string must not exceed MAX_PATH characters.

fInfoLevelId

[in] Specifies a **GET_FILEEX_INFO_LEVELS** enumeration type that gives the set of attribute information to obtain.

lpFileInformation

[out] 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**.

Remarks

The **GetFileAttributes** function retrieves a set of FAT-style attribute information. **GetFileAttributesEx** can obtain other sets of file or directory attribute information. Currently, **GetFileAttributeEx** retrieves a set of standard attributes that is a superset of the FAT-style attribute information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetFileAttributes, GET_FILEEX_INFO_LEVELS, SetFileAttributes

---

1.138 GetFileInformationByHandle

The **GetFileInformationByHandle** function retrieves file information for a specified file.

```
GetFileInformationByHandle: procedure
(
        hFile:              dword;
    var lpFileInformation:  BY_HANDLE_FILE_INFORMATION
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetFileInformationByHandle@8" );
```

Parameters

hFile

[in] Handle to the file for which to obtain information.

This handle should not be a pipe handle. The **GetFileInformationByHandle** function does not work with pipe handles.

lpFileInformation

[out] Pointer to a **BY_HANDLE_FILE_INFORMATION** structure that receives the file information. The structure can be used in subsequent calls to **GetFileInformationByHandle** to refer to the information about the file.

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

Depending on the underlying network components of the operating system and the type of server connected to, the **GetFileInformationByHandle** function may fail, return partial information, or full information for the given file. In general, you should not use **GetFileInformationByHandle** unless your application is intended to be run on a limited set of operating system configurations.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, BY_HANDLE_FILE_INFORMATION

1.139 GetFileSize

The **GetFileSize** function retrieves the size of a specified file.

This function stores the file size in a **DWORD** value. To retrieve a file size that is larger than a **DWORD** value, use the **GetFileSizeEx** function.

```
GetFileSize: procedure
(
        hFile:          dword;
    var lpFileSizeHigh: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetFileSize@8" );
```

Parameters

hFile

[in] Handle to the file whose size is to be returned. This handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

lpFileSizeHigh

[out] Pointer to the variable where the high-order word of the file size is returned. This parameter can be NULL if the application does not require the high-order word.

Return Values

If the function succeeds, the return value is the low-order doubleword of the file size, and, if *lpFileSizeHigh* is non-NULL, the function puts the high-order doubleword of the file size into the variable pointed to by that parameter.

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

If the function fails and *lpFileSizeHigh* is non-NULL, the return value is INVALID_FILE_SIZE and **GetLastError** will return a value other than NO_ERROR.

Remarks

You cannot use the **GetFileSize** function with a handle of a nonseeking device such as a pipe or a communications device. To determine the file type for *hFile*, use the **GetFileType** function.

The **GetFileSize** function retrieves the uncompressed size of a file. Use the **GetCompressedFileSize** function to obtain the compressed size of a file.

Note that if the return value is INVALID_FILE_SIZE and *lpFileSizeHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed. The following sample code illustrates this point:

```
// Case One: calling the function with
//           lpFileSizeHigh == NULL

// Try to obtain hFile's size
dwSize = GetFileSize (hFile, NULL) ;
```

```
// If we failed ...
if (dwSize == INVALID_FILE_SIZE)
{
    // Obtain the error code.
    dwError = GetLastError() ;

    // Deal with that failure.
    .
    .
    .

    } // End of error handler


//
// Case Two: calling the function with
//           lpFileSizeHigh != NULL

// Try to obtain hFile's huge size.
dwSizeLow = GetFileSize (hFile, & dwSizeHigh) ;

// If we failed ...
if (dwSizeLow == INVALID_FILE_SIZE
    &&
    (dwError = GetLastError()) != NO_ERROR )
{
    // Deal with that failure.
    .
    .
    .

    } // End of error handler.
```

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetCompressedFileSize, GetFileSizeEx, GetFileType

---

1.140 GetFileTime

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

```
    GetFileTime: procedure
    (
            hFile:              dword;
        var lpCreationTime:     FILETIME;
        var lpLastAccessTime:   FILETIME;
        var lpLastWriteTime:    FILETIME
    );
        stdcall;
        returns( "eax" );
        external( "__imp__GetFileTime@16" );
```

Parameters

hFile

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

lpCreationTime

[out] Pointer 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

[out] Pointer 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

[out] Pointer 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 NTFS file systems support the file creation, last access, and last write time values.

Windows NT/2000: If you rename or delete a file, then restore it shortly thereafter, Windows NT searches the cache for file information to restore. Cached information includes its short/long name pair and creation time.

Note Not all file systems can record creation and last access time and not all file systems record them in the same manner. For example, on Windows NT FAT, create time has a resolution of 10 milliseconds, write time has a resolution of 2 seconds, and access time has a resolution of 1 day (really, the access date). On NTFS, access time has a resolution of 1 hour. Therefore, **GetFileTime** may not return the same file time information set using the **SetFileTime** function. Furthermore, FAT records times on disk in local time. However, NTFS records times on disk in UTC, so it is not affected by changes in time zone or daylight saving time.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, GetFileSize, GetFileType, SetFileTime

---

1.141 GetFileType

The **GetFileType** function retrieves the file type for the specified file.

```
GetFileType: procedure
(
    hFile:dword
```

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

Parameters

hFile

[in] Handle to an open file.

Return Values

The return value is one of the following values.

Value	Meaning
FILE_TYPE_UNKNOWN	The type of the specified file is unknown.
FILE_TYPE_DISK	The specified file is a disk file.
FILE_TYPE_CHAR	The specified file is a character file, typically an LPT device or a console.
FILE_TYPE_PIPE	The specified file is either a named or anonymous pipe.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetFileSize, GetFileTime

---

1.142 GetFullPathName

The **GetFullPathName** function retrieves the full path and file name of a specified file.

```
GetFullPathName: procedure
(
        lpFileName:     string;
        nBufferLength:  dword;
    var lpBuffer:       var;
    var lpFilePart:     var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetFullPathNameA@16" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies a valid file name. This string can use either short (the 8.3 form) or long file names.

nBufferLength

[in] Specifies the size, in **TCHARs**, of the buffer for the drive and path.

lpBuffer

[out] Pointer to a buffer that receives the null-terminated string for the name of the drive and path.

lpFilePart

[out] Pointer to a buffer that receives the address (in *lpBuffer*) of the final file name component in the path.

Return Values

If the **GetFullPathName** function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpBuffer*, not including the terminating null character.

If the *lpBuffer* buffer is too small, the return value is the size of the buffer, in **TCHARs**, required to hold the path.

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

Remarks

GetFullPathName merges the name of the current drive and directory with the specified file name to determine the full path and file name of the specified file. It also calculates the address of the file name portion of the full path and file name. This function does not verify that the resulting path and file name are valid or that they refer to an existing file on the associated volume.

GetFullPathName does no conversion of the specified file name, *lpFileName*. If the specified file name exists, you can use **GetLongPathName** and **GetShortPathName** to convert to long and short path names, respectively.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, File I/O Functions, GetLongPathName, GetShortPathName, GetTempPath, SearchPath

---

1.143 GetHandleInformation

The **GetHandleInformation** function retrieves certain properties of an object handle.

```
GetHandleInformation: procedure
(
        hObject:    dword;
    var lpdwFlags:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetHandleInformation@8" );
```

Parameters

hObject

[in] Specifies a handle to an object. The **GetHandleInformation** function obtains information about this object handle.

You can specify a handle to one of the following types of objects: access token, event, file, file mapping, job, mailslot, mutex, pipe, printer, process, registry key, semaphore, serial communication device, socket, thread, or waitable timer.

Windows 2000: This parameter can also be a handle to a console input buffer or a console screen buffer.

lpdwFlags

[out] Pointer to a variable that receives a set of bit flags that specify properties of the object handle. The following flags are defined:

Value	Meaning
HANDLE_FLAG_INHERIT	If this flag is set, a child process created with the *bInheritHandles* parameter of **CreateProcess** set to TRUE will inherit the object handle.
HANDLE_FLAG_PROTECT_FROM_CLOSE	If this flag is set, calling the **CloseHandle** function will not close the object handle.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Handles and Objects Overview, Handle and Object Functions, CloseHandle, CreateProcess, SetHandleInformation

---

1.144 GetLargestConsoleWindowSize

The **GetLargestConsoleWindowSize** function retrieves the size of the largest possible console window, based on the current font and the size of the display.

```
GetLargestConsoleWindowSize: procedure
(
    hConsoleOutput:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetLargestConsoleWindowSize@4" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer.

Return Values

If the function succeeds, the return value (in EAX) is a **COORD** structure that specifies the number of character cell rows (**X** member in AX) and columns (**Y** member in H.O. word of EAX) in the largest possible console window. Oth-

erwise, the members of the structure are zero.

To get extended error information, call **GetLastError**.

Remarks

The function does not take into consideration the size of the screen buffer, which means that the window size returned may be larger than the size of the screen buffer. The **GetConsoleScreenBufferInfo** function can be used to determine the maximum size of the console window, given the current screen buffer size, the current font, and the display size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, COORD, GetConsoleScreenBufferInfo, SetConsoleWindowInfo

---

1.145 GetLastError

The **GetLastError** function retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

```
GetLastError: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetLastError@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the calling thread’s last-error code value. Functions set this value by calling the **SetLastError** function. The **Return Value** section of each reference page notes the conditions under which the function sets the last-error code.

Windows 95/98: Because **SetLastError** is a 32-bit function only, Win32 functions that are actually implemented in 16-bit code do not set the last-error code. You should ignore the last-error code when you call these functions. They include window management functions, GDI functions, and Multimedia functions.

Remarks

To obtain an error string for system error codes, use the **FormatMessage** function. For a complete list of error codes, see Error Codes.

You should call the **GetLastError** function immediately when a function's return value indicates that such a call will return useful data. That is because some functions call **SetLastError**(0) when they succeed, wiping out the error code set by the most recently failed function.

Most functions in the Win32 API that set the thread's last error code value set it when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference page.

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your application, set this bit to one. That

indicates that the error code has been defined by an application, and ensures that your error code does not conflict with any error codes defined by the system.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf.
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions, FormatMessage, SetLastError, SetLastErrorEx

---

1.146 GetLocalTime

The **GetLocalTime** function retrieves the current local date and time.

```
GetLocalTime: procedure
(
    var lpSystemTime: SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetLocalTime@4" );
```

Parameters

lpSystemTime

[out] Pointer to a **SYSTEMTIME** structure to receive the current local date and time.

Return Values

This function does not return a value.

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetSystemTime, SetLocalTime, SYSTEMTIME

1.147 GetLocaleInfo

The **GetLocaleInfo** function retrieves information about a locale.

```
GetLocaleInfo: procedure
(
        Locale:     LCID;
```

```
        LCType:     LCTYPE;
    var lpLCData:   var;
        cchData:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetLocaleInfoA@16" );
```

Parameters

Locale

[in] Specifies the locale to retrieve information for. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

LCType

[in] Specifies the type of locale information to be retrieved, by using an **LCTYPE** constant. For a list of **LCTYPE** constants, see Locale Information.

Note that only one **LCTYPE** constant may be specified per call, except that an application may use the binary-OR operator to combine LOCALE_NOUSEROVERRIDE or LOCALE_USE_CP_ACP with any other **LCTYPE** value.

If LOCALE_NOUSEROVERRIDE is combined with another value, the function bypasses user overrides, and returns the system default value for the requested LCID. The information is retrieved from the locale database, even if the LCID is the current one and the user has changed some of the values in Control Panel. If this flag is not specified, the values in Win.ini take precedence over the database settings when getting values for the current system default locale.

lpLCData

[out] Pointer to a buffer that receives the requested data.This pointer is not used if *cchData* is zero.

cchData

[in] Specifies the size, in **TCHARs**, of the *lpLCData* buffer. If *cchData* is zero, the function returns the number of bytes or characters required to hold the information, including the terminating null character, and the buffer pointed to by *lpLCData* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the destination buffer. If the *cchData* parameter is zero, the return value is the number of bytes or characters required to hold the locale information.

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

Remarks

The **GetLocaleInfo** function always retrieves information in text format. If the information is a numeric value, the function converts the number to text using decimal notation.

The LOCALE_FONTSIGNATURE parameter will return a non-NULL terminated string. In all other cases, the string

is NULL terminated.

The ANSI string returned by the ANSI version of this function is translated from Unicode to ANSI based on the default ANSI code page for the LCID. However, if LOCALE_USE_CP_ACP is specified, the translation is based on the system default–ANSI code page.

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only LCID. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetStringTypeA, GetStringTypeEx, GetStringTypeW, GetSystemDefaultLCID, GetUserDefaultLCID, LCTYPE Constants, SetLocaleInfo, MAKELCID

1.148 GetLogicalDriveStrings

The **GetLogicalDriveStrings** function fills a buffer with strings that specify valid drives in the system.

```
GetLogicalDriveStrings: procedure
(
        nBufferLength:  dword;
    var lpBuffer:       var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetLogicalDriveStringsA@8" );
```

Parameters

nBufferLength

[in] Specifies the maximum size, in characters, of the buffer pointed to by *lpBuffer*. This size does not include the terminating null character.

lpBuffer

[out] Pointer to a buffer that receives a series of null-terminated strings, one for each valid drive in the system, that end with a second null character. The following example shows the buffer contents with <null> representing the terminating null character.

```
c:\<null>d:\<null><null>
```

Return Values

If the function succeeds, the return value is the length, in characters, of the strings copied to the buffer, not including the terminating null character. Note that an ANSI-ASCII null character uses one byte, but a Unicode null character uses two bytes.

If the buffer is not large enough, the return value is greater than *nBufferLength*. It is the size of the buffer required to hold the drive strings.

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

Remarks

Each string in the buffer may be used wherever a root directory is required, such as for the **GetDriveType** and **Get-**

DiskFreeSpace functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetDriveType, GetDiskFreeSpace, GetLogicalDrives

---

1.149 GetLogicalDrives

The **GetLogicalDrives** function retrieves a bitmask representing the currently available disk drives.

```
GetLogicalDrives: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetLogicalDrives@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a bitmask representing the currently available disk drives. Bit position 0 (the least-significant bit) is drive A, bit position 1 is drive B, bit position 2 is drive C, and so on.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetLogicalDriveStrings

---

1.150 GetLongPathName

The **GetLongPathName** function converts the specified path to its long form. If no long path is found, this function simply returns the specified name.

```
GetLongPathName: procedure
(
        lpszShortPath:  string;
    var lpszLongPath:   var;
        cchBuffer:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetLongPathNameA@12" );
```

Parameters

lpszShortPath

[in] Pointer to a null-terminated path to be converted.

Windows 2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 98: This string must not exceed MAX_PATH characters.

lpszLongPath

[out] Pointer to the buffer to receive the long path. You can use the same buffer you used for the *lpszShortPath* parameter.

cchBuffer

[in] Specifies the size of the buffer, in **TCHARs**.

Return Values

If the function succeeds, the return value is the length of the string copied to the *lpszLongPath* parameter, in **TCHARs**. This length does not include the terminating null character.

If *lpszLongPath* is too small, the function returns the size, in **TCHARs**, of the buffer required to hold the long path.

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

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetFullPathName, GetShortPathName

---

1.151 GetMailslotInfo

The **GetMailslotInfo** function retrieves information about the specified mailslot.

```
GetMailslotInfo: procedure
(
        hMailslot:          dword;
    var lpMaxMessageSize:   dword;
    var lpNextSize:         dword;
    var lpMessageCount:     dword;
    var lpReadTimeout:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetMailslotInfo@20" );
```

Parameters

hMailslot

[in] Handle to a mailslot. The **CreateMailslot** function must create this handle.

lpMaxMessageSize

[in] Pointer to a buffer specifying the maximum message size, in bytes, allowed for this mailslot, when the function returns. This value can be greater than or equal to the value specified in the *cbMaxMsg* parameter of the **CreateMailslot** function that created the mailslot. This parameter can be NULL.

lpNextSize

[in] Pointer to a buffer specifying the size, in bytes, of the next message, when the function returns. The following value has special meaning.

Value	Meaning
MAILSLOT_NO_MESSAGE	There is no next message.

This parameter can be NULL.

lpMessageCount

[in] Pointer to a buffer specifying the total number of messages waiting to be read, when the function returns. This parameter can be NULL.

lpReadTimeout

[in] Pointer to a buffer specifying the amount of time, in milliseconds, a read operation can wait for a message to be written to the mailslot before a time-out occurs. This parameter is filled in when the function returns. This parameter can be NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Mailslots Overview, Mailslot Functions, CreateMailslot, SetMailslotInfo

---

1.152 GetModuleFileName

The **GetModuleFileName** function retrieves the full path and file name for the file containing the specified module.

Windows 95/98: The **GetModuleFilename** function retrieves long file names when an application's version number is greater than or equal to 4.00 and the long file name is available. Otherwise, it returns only 8.3 format file names.

```
        GetModuleFileName: procedure
    (
            hModule:    dword;
        var lpFilename: var;
            nSize:      dword
    );
        stdcall;
        returns( "eax" );
        external( "__imp__GetModuleFileNameA@12" );
```

Parameters

hModule

[in] Handle to the module whose file name is being requested. If this parameter is NULL, **GetModuleFileName** returns the path for the file containing the current process.

lpFilename

[out] Pointer to a buffer that receives the path and file name of the specified module.

nSize

[in] Specifies the length, in **TCHARs**, of the *lpFilename* buffer. If the length of the path and file name exceeds this limit, the string is truncated.

Return Values

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to the buffer.

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

Remarks

If a DLL is loaded in two processes, its file name in one process may differ in case from its file name in the other process.

For the ANSI version of the function, the number of **TCHARs** is the number of bytes; for the Unicode version, it is the number of characters.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, GetModuleHandle, LoadLibrary

---

1.153 GetModuleHandle

The **GetModuleHandle** function retrieves a module handle for the specified module if the file has been mapped into the address space of the calling process.

```
GetModuleHandle: procedure
(
    lpModuleName:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetModuleHandleA@4" );
```

Parameters

lpModuleName

[in] Pointer to a null-terminated string that contains the name of the module (either a .dll or .exe file). If the file name extension is omitted, the default library extension .dll is appended. The file name string can include a trailing point character (.) to indicate that the module name has no extension. The string does not have to specify a path. When specifying a path, be sure to use backslashes (\), not forward slashes (/). The name is compared (case

independently) to the names of modules currently mapped into the address space of the calling process.

If this parameter is NULL, **GetModuleHandle** returns a handle to the file used to create the calling process.

Return Values

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

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

Remarks

The returned handle is not global, inheritable, or duplicative, and it cannot be used by another process.

The handles returned by **GetModuleHandle** and **LoadLibrary** can be used in the same functions for example, **GetProcAddress, FreeLibrary**, or **LoadResource**. The difference between the two functions involves the reference count. **LoadLibrary** maps the module into the address space of the calling process, if necessary, and increments the module's reference count, if it is already mapped. **GetModuleHandle**, however, returns the handle to a mapped module without incrementing its reference count.

Note that the reference count is used in **FreeLibrary** to determine whether to unmap the function from the address space of the process. For this reason, use care when using a handle returned by **GetModuleHandle** in a call to **FreeLibrary** because doing so can cause a dynamic-link library (DLL) module to be unmapped prematurely.

This function must also be used carefully in a multithreaded application. There is no guarantee that the module handle remains valid between the time this function returns the handle and the time it is used by another function. For example, a thread might retrieve a module handle by calling **GetModuleHandle.** Before the thread uses the handle in another function, a second thread could free the module and the system could load another module, giving it the same handle as the module that was recently freed. The first thread would then be left with a module handle that refers to a module different than the one intended.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, FreeLibrary, GetModuleFileName, GetProcAddress, LoadLibrary, LoadResource

---

1.154 GetNamedPipeHandleState

The **GetNamedPipeHandleState** function retrieves information about a specified named pipe. The information returned can vary during the lifetime of an instance of the named pipe.

```
GetNamedPipeHandleState: procedure
(
        hNamedPipe:             dword;
    var lpState:                var;
    var lpCurInstances:         var;
    var lpMaxCollectionCount:   var;
    var lpCollectDataTimeout:   var;
    var lpUserName:             var;
        nMaxUserNameSize:       dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetNamedPipeHandleStateA@28" );
```

Parameters

hNamedPipe

[in] Handle to the named pipe for which information is wanted. The handle must have GENERIC_READ access to the named pipe.

Windows NT/2000: This parameter can also be a handle to an anonymous pipe, as returned by the **`CreatePipe`** function.

lpState

[out] Pointer to a variable that indicates the current state of the handle. This parameter can be NULL if this information is not needed. Either or both of the following values can be specified.

Value	Meaning
PIPE_NOWAIT	The pipe handle is in nonblocking mode. If this flag is not specified, the pipe handle is in blocking mode.
PIPE_READMODE_MESSAGE	The pipe handle is in message-read mode. If this flag is not specified, the pipe handle is in byte-read mode.

lpCurInstances

[out] Pointer to a variable that receives the number of current pipe instances. This parameter can be NULL if this information is not required.

lpMaxCollectionCount

[out] Pointer to a variable that receives the maximum number of bytes to be collected on the client's computer before transmission to the server. This parameter must be NULL if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same computer. This parameter can be NULL if this information is not required.

lpCollectDataTimeout

[out] Pointer to a variable that receives the maximum time, in milliseconds, that can pass before a remote named pipe transfers information over the network. This parameter must be NULL if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same computer. This parameter can be NULL if this information is not required.

lpUserName

[out] Pointer to a buffer that receives the null-terminated string containing the user name string associated with the client application. The server can only retrieve this information if the client opened the pipe with SECURITY_IMPERSONATION access.

This parameter must be NULL if the specified pipe handle is to the client end of a named pipe. This parameter can be NULL if this information is not required.

nMaxUserNameSize

[in] Specifies the size, in **TCHARs**, of the buffer specified by the *lpUserName* parameter. This parameter is ignored if *lpUserName* is NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **GetNamedPipeHandleState** function returns successfully even if all of the pointers passed to it are NULL.

To set the pipe handle state, use the **SetNamedPipeHandleState** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Pipes Overview, Pipe Functions, SetNamedPipeHandleState

1.155 GetNamedPipeInfo

The **GetNamedPipeInfo** function retrieves information about the specified named pipe.

```
GetNamedPipeInfo: procedure
(
        hNamedPipe:         dword;
    var lpFlags:            dword;
    var lpOutBufferSize:    dword;
    var lpInBufferSize:     dword;
    var lpMaxInstances:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetNamedPipeInfo@20" );
```

Parameters

hNamedPipe

[in] Handle to the named pipe instance. The handle must have GENERIC_READ access to the named pipe.

Windows NT/2000: This parameter can also be a handle to an anonymous pipe, as returned by the **CreatePipe** function.

lpFlags

[in] Pointer to a variable that indicates the type of the named pipe. This parameter can be NULL if this information is not required. Otherwise, use the following values.

Value	Meaning
PIPE_CLIENT_END	The handle refers to the client end of a named pipe instance. This is the default.
PIPE_SERVER_END	The handle refers to the server end of a named pipe instance. If this value is not specified, the handle refers to the client end of a named pipe instance.
PIPE_TYPE_BYTE	The named pipe is a byte pipe. This is the default.
PIPE_TYPE_MESSAGE	The named pipe is a message pipe. If this value is not specified, the pipe is a byte pipe.

lpOutBufferSize

[out] Pointer to a variable that receives the size, in bytes, of the buffer for outgoing data. If the buffer size is zero, the buffer is allocated as needed. This parameter can be NULL if this information is not required.

lpInBufferSize

[out] Pointer to a variable that receives the size, in bytes, of the buffer for incoming data. If the buffer size is zero, the buffer is allocated as needed. This parameter can be NULL if this information is not required.

lpMaxInstances

[out] Pointer to a variable that receives the maximum number of pipe instances that can be created. If the variable is set to PIPE_UNLIMITED_INSTANCES, the number of pipe instances that can be created is limited only by the availability of system resources. This parameter can be NULL if this information is not required.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CreateNamedPipe, GetNamedPipeHandleState

---

1.156 GetNumberFormat

The **GetNumberFormat** function formats a number string as a number string customized for a specified locale.

```
GetNumberFormat: procedure
(
        Locale:     LCID;
        dwFlags:    dword;
        lpValue:    string;
    var lpFormat:   NUMBERFMT;
    var lpNumberStr:var;
        cchNumber:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetNumberFormatA@24" );
```

Parameters

Locale

[in] Specifies the locale for which the number string is to be formatted. If *lpFormat* is NULL, the function formats the string according to the number format for this locale. If *lpFormat* is not NULL, the function uses the locale only for formatting information not specified in the **NUMBERFMT** structure (for example, the locale's string value for the negative sign).

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.

LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Controls the operation of the function. If *lpFormat* is non-NULL, this parameter must be zero.

If *lpFormat* is NULL, you can specify LOCALE_NOUSEROVERRIDE to format the string using the system default number format for the specified locale; or you can specify zero to format the string using any user overrides to the locale's default number format.

lpValue

[in] Pointer to a null-terminated string containing the number string to format.

This string can only contain the following characters:

Characters '0' through '9'.

One decimal point (dot) if the number is a floating-point value.

A minus sign in the first character position if the number is a negative value.

All other characters are invalid. The function returns an error if the string pointed to by *lpValue* deviates from these rules.

lpFormat

[in] Pointer to a **NUMBERFMT** structure that contains number formatting information. All members in the structure pointed to by *lpFormat* must contain appropriate values.

If *lpFormat* is NULL, the function uses the number format of the specified locale.

lpNumberStr

[out] Pointer to a buffer that receives the formatted number string.

cchNumber

[in] Specifies the size, in **TCHARs**, of the *lpNumberStr* buffer. If *cchNumber* is zero, the function returns the number of bytes or characters required to hold the formatted number string, and the buffer pointed to by *lpNumberStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the buffer pointed to by *lpNumberStr*, or if the *cchNumber* parameter is zero, the number of bytes or characters required to hold the formatted number string. The count includes the terminating null.

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

Remarks

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only locale. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetCurrencyFormat, NUMBERFMT

1.157 GetNumberOfConsoleInputEvents

The **GetNumberOfConsoleInputEvents** function retrieves the number of unread input records in the console's input buffer.

```
GetNumberOfConsoleInputEvents: procedure
(
        hConsoleInput:      dword;
    var lpcNumberOfEvents:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetNumberOfConsoleInputEvents@8" );
```

Parameters

hConsoleInput

[in] Handle to the console input buffer. The handle must have GENERIC_READ access.

lpcNumberOfEvents

[out] Pointer to a variable that receives the number of unread input records in the console's input buffer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **GetNumberOfConsoleInputEvents** function reports the total number of unread input records in the input buffer, including keyboard, mouse, and window-resizing input records. Processes using the **ReadFile** or **ReadConsole** function can only read keyboard input. Processes using the **ReadConsoleInput** function can read all types of input records.

A process can specify a console input buffer handle in one of the wait functions to determine when there is unread console input. When the input buffer is not empty, the state of a console input buffer handle is signaled.

To read input records from a console input buffer without affecting the number of unread records, use the **PeekConsoleInput** function. To discard all unread records in a console's input buffer, use the **FlushConsoleInputBuffer** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, FlushConsoleInputBuffer, PeekConsoleInput, ReadConsole, ReadConsoleInput, ReadFile

1.158 GetNumberOfConsoleMouseButtons

The **GetNumberOfConsoleMouseButtons** function retrieves the number of buttons on the mouse used by the current console.

```
GetNumberOfConsoleMouseButtons: procedure
(
    var lpNumberOfMouseButtons: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetNumberOfConsoleMouseButtons@4" );
```

Parameters

lpNumberOfMouseButtons

[out] Pointer to a variable that receives the number of mouse buttons.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

When a console receives mouse input, an **INPUT_RECORD** structure containing a **MOUSE_EVENT_RECORD** structure is placed in the console's input buffer. The **dwButtonState** member of **MOUSE_EVENT_RECORD** has a bit indicating the state of each mouse button. The bit is 1 if the button is down and 0 if the button is up. To determine the number of bits that are significant, use **GetNumberOfConsoleMouseButtons**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, ReadConsoleInput, INPUT_RECORD, MOUSE_EVENT_RECORD

1.159 GetOEMCP

The **GetOEMCP** function retrieves the current original equipment manufacturer (OEM) code-page identifier for the system.

```
GetOEMCP: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetOEMCP@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the current OEM code-page identifier for the system or a default identifier if no code page is current.

Remarks

The following are the OEM code-page identifiers.

Identifier	Meaning
437	MS-DOS United States
708	Arabic (ASMO 708)
709	Arabic (ASMO 449+, BCON V4)
710	Arabic (Transparent Arabic)
720	Arabic (Transparent ASMO)
737	Greek (formerly 437G)
775	Baltic
850	MS-DOS Multilingual (Latin I)
852	MS-DOS Slavic (Latin II)
855	IBM Cyrillic (primarily Russian)
857	IBM Turkish
860	MS-DOS Portuguese
861	MS-DOS Icelandic
862	Hebrew
863	MS-DOS Canadian-French
864	Arabic
865	MS-DOS Nordic
866	MS-DOS Russian
869	IBM Modern Greek
874	Thai
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1361	Korean (Johab)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetACP

1.160 GetOverlappedResult

The **GetOverlappedResult** function retrieves the results of an overlapped operation on the specified file, named pipe, or communications device.

Windows 95/98: This function works only on communications devices or on files opened by using the **DeviceIoControl** function.

```
GetOverlappedResult: procedure
(
        hFile:                      dword;
    var lpOverlapped:               OVERLAPPED;
    var lpNumberOfBytesTransferred: dword;
        bWait:                      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetOverlappedResult@16" );
```

Parameters

hFile

[in] Handle to the file, named pipe, or communications device. This is the same handle that was specified when the overlapped operation was started by a call to the **ReadFile, WriteFile, ConnectNamedPipe, TransactNamedPipe, DeviceIoControl,** or **WaitCommEvent** function.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure that was specified when the overlapped operation was started.

lpNumberOfBytesTransferred

[out] Pointer to a variable that receives the number of bytes that were actually transferred by a read or write operation. For a **TransactNamedPipe** operation, this is the number of bytes that were read from the pipe. For a **DeviceIoControl** operation, this is the number of bytes of output data returned by the device driver. For a **ConnectNamedPipe** or **WaitCommEvent** operation, this value is undefined.

bWait

[in] Specifies whether the function should wait for the pending overlapped operation to be completed. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the **GetLastError** function returns ERROR_IO_INCOMPLETE.

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 results reported by the **GetOverlappedResult** function are those of the specified handle's last overlapped operation to which the specified **OVERLAPPED** structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns FALSE, and the **GetLastError** function returns ERROR_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the **hEvent** member of the **OVERLAPPED** structure to the nonsignaled state. Then when the pending opera-

tion has been completed, the system sets the event object to the signaled state.

Specify a manual-reset event object in the **OVERLAPPED** structure. If an auto-reset event object is used, the event handle must not be specified in any other wait operation in the interval between starting the overlapped operation and the call to **GetOverlappedResult**. For example, the event object is sometimes specified in one of the wait functions to wait for the operation's completion. When the wait function returns, the system sets an auto-reset event's state to nonsignaled, and a subsequent call to **GetOverlappedResult** with the *bWait* parameter set to TRUE causes the function to be blocked indefinitely.

If the *bWait* parameter is TRUE, **GetOverlappedResult** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state.

Windows 95/98: If *bWait* is TRUE, the **hEvent** member of the **OVERLAPPED** structure must not be NULL.

Windows NT/2000: If the **hEvent** member of the **OVERLAPPED** structure is NULL, the system uses the state of the *hFile* handle to signal when the operation has been completed. Use of file, named pipe, or communications-device handles for this purpose is discouraged. It is safer to use an event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CancelIo, ConnectNamedPipe, CreateEvent, DeviceIoControl, GetLastError, OVERLAPPED, ReadFile, TransactNamedPipe, WaitCommEvent, WriteFile

---

1.161 GetPriorityClass

The **GetPriorityClass** function retrieves the priority class for the specified process. This value, together with the priority value of each thread of the process, determines each thread's base priority level.

```
GetPriorityClass: procedure
(
    hProcess:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPriorityClass@4" );
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have PROCESS_QUERY_INFORMATION access. For more information, see Process Security and Access Rights.

Return Values

If the function succeeds, the return value is the priority class of the specified process.

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

The process's priority class is one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.
IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Remarks

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base priority levels for each combination of priority class and thread priority value, see the **`SetPriorityClass`** function.

Windows NT 4.0 and earlier: Priority class is maintained by the Windows subsystem (csrss), so only Windows-based application have a priority class that can be queried.

Windows 2000: Priority class is maintained by the executive, so all processes have a priority class that can be queried.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetThreadPriority, SetPriorityClass, SetThreadPriority

1.162 GetPrivateProfileInt

The **GetPrivateProfileInt** function retrieves an integer associated with a key in the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetPrivateProfileInt: procedure
(
    lpAppName:  string;
    lpKeyName:  string;
    nDefault:   dword;
    lpFileName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPrivateProfileIntA@16" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string specifying the name of the section in the initialization file.

lpKeyName

[in] Pointer to the null-terminated string specifying the name of the key 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

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

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return Values

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/2000: 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:

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**

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetProfileInt, WritePrivateProfileString

1.163 GetPrivateProfileSection

The **GetPrivateProfileSection** function retrieves all the keys and values for the specified section of an initialization file.

Windows 95: The specified profile section must not exceed 32K.

Windows NT/2000: The specified profile section has no size limit.

Note This function is provided only for compatibility with 16-bit applications written for Windows. Win32-based applications should store initialization information in the registry.

```
GetPrivateProfileSection: procedure
(
        lpAppName:          string;
    var lpReturnedString:   var;
        nSize:              dword;
        lpFileName:         string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPrivateProfileSectionA@16" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string specifying the name of the section in the initialization file.

lpReturnedString

[out] Pointer 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

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpReturnedString* parameter.

Windows 95: The maximum buffer size is 32,767 characters.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the initialization file. If this parameter does not contain a full path to the file, the system 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/2000: 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 follow-

ing 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:

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**

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.

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.

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.

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.

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.

Windows NT/2000: Comments (any line that starts with a semicolon) are stripped out and not returned in the *lpReturnedString* buffer.

Windows 95/98: The *lpReturnedString* buffer receives a copy of the entire section, including comments.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetProfileSection, WritePrivateProfileSection

1.164 GetPrivateProfileSectionNames

The **GetPrivateProfileSectionNames** function retrieves the names of all sections in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetPrivateProfileSectionNames: procedure
(
    var lpszReturnBuffer:   var;
        nSize:              dword;
        lpFileName:         string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPrivateProfileSectionNamesA@12" );
```

Parameters

lpszReturnBuffer

[out] Pointer to a buffer that receives the section names associated with the named file. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpszReturnBuffer* parameter.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the initialization file. If this parameter is NULL, the function searches the Win.ini file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return Values

The return value specifies the number of characters copied to the specified buffer, not including the terminating null character. If the buffer is not large enough to contain all the section names associated with the specified initialization file, the return value is equal to the length specified by *nSize* minus two.

Remarks

This operation is atomic; no updates to the initialization file are allowed while the section names are being copied to the buffer.

Calls to profile functions might be mapped to the registry instead of to the initialization files. When the operation has been mapped, the **GetPrivateProfileSectionNames** 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileSection, WritePrivateProfileSection

---

1.165 GetPrivateProfileString

The **GetPrivateProfileString** function retrieves a string from the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetPrivateProfileString: procedure
(
        lpAppName:          string;
        lpKeyName:          string;
        lpDefault:          string;
    var lpReturnedString:   var;
        nSize:              dword;
        lpFileName:         string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPrivateProfileStringA@24" );
```

Parameters

lpAppName

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

[in] Pointer to the null-terminated string specifying the name of the key 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

[in] 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, the system strips any trailing blanks by inserting a null character into the *lpDefault* string before copying it to the *lpReturnedString* buffer.

Windows NT/2000: The system 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

[out] Pointer to the buffer that receives the retrieved string.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpReturnedString* parameter.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return Values

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/2000: 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetProfileString, WritePrivateProfileString

---

1.166 GetPrivateProfileStruct

The **GetPrivateProfileStruct** function retrieves the data associated with a key in the specified section of an initialization file. As it retrieves the data, the function calculates a checksum and compares it with the checksum calculated by the **WritePrivateProfileStruct** function when the data was added to the file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetPrivateProfileStruct: procedure
(
        lpszSection:    string;
        lpszKey:        string;
    var lpStruct:       var;
        uSizeStruct:    dword;
        szFile:         string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetPrivateProfileStructA@20" );
```

Parameters

lpszSection

[in] Pointer to a null-terminated string specifying the name of the section in the initialization file.

lpszKey

[in] Pointer to the null-terminated string specifying the name of the key whose data is to be retrieved.

lpStruct

[out] Pointer to the buffer that receives the data associated with the file, section, and key names.

uSizeStruct

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpStruct* parameter.

szFile

[in] Pointer to a null-terminated string that specifies the name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

A section in the initialization file must have the following form:

[*section***]**

*key***=***data*

.

.

.

Calls to private profile functions might be mapped to the registry instead of to the specified initialization files. 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 **GetPrivateProfileStruct** 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, WritePrivateProfileStruct

---

1.167 GetProcAddress

The **GetProcAddress** function retrieves the address of the specified exported dynamic-link library (DLL) function.

```
GetProcAddress: procedure
(
    hModule:    dword;
    lpProcName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcAddress@8" );
```

Parameters

hModule

[in] Handle to the DLL module that contains the function. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

lpProcName

[in] Pointer to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

Return Values

If the function succeeds, the return value is the address of the DLL's exported function.

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

Remarks

The **GetProcAddress** function is used to retrieve addresses of exported functions in DLLs.

The spelling and case of the function name pointed to by *lpProcName* must be identical to that in the **EXPORTS** statement of the source DLL's module-definition (.DEF) file. The exported names of Win32 API functions may differ from the names you use when calling these functions in your code. This difference is hidden by macros used in the SDK header files. For more information, see Win32 Function Prototypes.

The *lpProcName* parameter can identify the DLL function by specifying an ordinal value associated with the function in the **EXPORTS** statement. **GetProcAddress** verifies that the specified ordinal is in the range 1 through the highest ordinal value exported in the .DEF file. The function then uses the ordinal as an index to read the function's address from a function table. If the .DEF file does not number the functions consecutively from 1 to *N* (where *N* is the number of exported functions), an error can occur where **GetProcAddress** returns an invalid, non-NULL address, even though there is no function with the specified ordinal.

In cases where the function may not exist, the function should be specified by name rather than by ordinal value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, FreeLibrary, GetModuleHandle, LoadLibrary

---

1.168 GetProcessAffinityMask

The **GetProcessAffinityMask** function retrieves the process affinity mask for the specified process and the system affinity mask for the system.

A process affinity mask is a bit vector in which each bit represents the processors that a process is allowed to run on.

A system affinity mask is a bit vector in which each bit represents the processors that are configured into a system.

A process affinity mask is a proper subset of a system affinity mask. A process is only allowed to run on the processors configured into a system.

```
GetProcessAffinityMask: procedure
(
        hProcess:               dword;
    var lpProcessAffinityMask:  dword;
    var lpSystemAffinityMask:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessAffinityMask@12" );
```

Parameters

hProcess

[in] Handle to the process whose affinity mask is desired.

Windows NT/2000: This handle must have PROCESS_QUERY_INFORMATION access. For more information, see Process Security and Access Rights.

lpProcessAffinityMask

[out] Pointer to a variable that receives the affinity mask for the specified process.

lpSystemAffinityMask

[out] Pointer to a variable that receives the affinity mask for the system.

Return Values

If the function succeeds, the return value is nonzero.

Windows NT/2000: Upon success, the function sets the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the appropriate affinity masks.

Windows 95/98: Upon success, the function sets the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the value one.

If the function fails, the return value is zero, and the values of the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* are undefined. To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, SetProcessAffinityMask, SetThreadAffinityMask

---

1.169 GetProcessHeap

The **GetProcessHeap** function obtains a handle to the heap of the calling process. This handle can then be used in subsequent calls to the **HeapAlloc, HeapReAlloc, HeapFree**, and **HeapSize** functions.

```
GetProcessHeap: procedure;
    stdcall;
    returns( "eax" );
```

```
    external( "__imp__GetProcessHeap@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the calling process's heap.

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

Remarks

The **GetProcessHeap** function allows you to allocate memory from the process heap without having to first create a heap with the **HeapCreate** function, as shown in this example:

```
HeapAlloc(GetProcessHeap(), 0, dwBytes);
```

Note The handle obtained by calling this function should not be used in calls to the **HeapDestroy** function.

To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapAlloc, HeapCreate, HeapDestroy, HeapFree, HeapReAlloc, HeapSize

---

1.170 GetProcessHeaps

The **GetProcessHeaps** function obtains handles to all of the heaps that are valid for the calling process.

```
GetProcessHeaps: procedure
(
        NumberOfHeaps:  dword;
    var ProcessHeaps:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessHeaps@8" );
```

Parameters

NumberOfHeaps

[in] Specifies the maximum number of heap handles that can be stored into the buffer pointed to by *ProcessHeaps*.

ProcessHeaps

[out] Pointer to a buffer to receive an array of heap handles.

Return Values

The return value is the number of heap handles that are valid for the calling process.

If the return value is less than or equal to *NumberOfHeaps*, it is also the number of heap handles stored into the buffer pointed to by *ProcessHeaps*.

If the return value is greater than *NumberOfHeaps*, the buffer pointed to by *ProcessHeaps* is too small to hold all the valid heap handles of the calling process.The function will have stored no handles into that buffer. In this situation, use the return value to allocate a buffer that is large enough to receive the handles, and call the function again.

If the return value is zero, the function has failed, because every process has at least one valid heap, the process heap. To get extended error information, call **GetLastError**.

Remarks

Use the **GetProcessHeap** function to obtain a handle to the process heap of the calling process. The **GetProcessHeaps** function obtains a handle to that heap, plus handles to any additional private heaps created by calling the **HeapCreate** function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapCreate

---

1.171 GetProcessPriorityBoost

The **GetProcessPriorityBoost** function retrieves the priority boost control state of the specified process.

```
GetProcessPriorityBoost: procedure
(
        hProcess:               dword;
    var pDisablePriorityBoost:  boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessPriorityBoost@8" );
```

Parameters

hProcess

[in] Handle to the process. This handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see Process Security and Access Rights.

pDisablePriorityBoost

[out] Pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return Values

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, SetProcessPriorityBoost

---

1.172 GetProcessShutdownParameters

The **GetProcessShutdownParameters** function retrieves shutdown parameters for the currently calling process.

```
BOOL GetProcessShutdownParameters(
  LPDWORD lpdwLevel, // shutdown priority
  LPDWORD lpdwFlags  // shutdown flag
);
```

Parameters

lpdwLevel

[out] Pointer to a variable that receives the shutdown priority level. Higher levels shut down first. System level shutdown orders are reserved for system components. Higher numbers shut down first. Following are the level conventions.

Value	Meaning
000–0FF	System reserved last shutdown range.
100–1FF	Application reserved last shutdown range.
200–2FF	Application reserved "in between" shutdown range.
300–3FF	Application reserved first shutdown range.
400–4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

lpdwFlags

[out] Pointer to a variable that receives the shutdown flags. This parameter can be the following value.

Value	Meaning
SHUTDOWN_NORETRY	If this process takes longer than the specified timeout to shut down, do not display a retry dialog box for the user. Instead, just cause the process to directly exit.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

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

Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, SetProcessShutdownParameters

---

1.173 GetProcessTimes

The **GetProcessTimes** function retrieves timing information for the specified process.

```
GetProcessTimes: procedure
(
        hProcess:       dword;
    var lpCreationTime: FILETIME;
    var lpExitTime:     FILETIME;
    var lpKernelTime:   FILETIME;
    var lpUserTime:     FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessTimes@20" );
```

Parameters

hProcess

[in] Handle to the process whose timing information is sought. This handle must be created with PROCESS_QUERY_INFORMATION access. For more information, see Process Security and Access Rights.

lpCreationTime

[out] Pointer to a **FILETIME** structure that receives the creation time of the process.

lpExitTime

[out] Pointer to a **FILETIME** structure that receives the exit time of the process. If the process has not exited, the content of this structure is undefined.

lpKernelTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the process has executed in kernel mode. The time that each of the threads of the process has executed in kernel mode is determined, and then all of those times are summed together to obtain this value.

lpUserTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the process has executed in user mode. The time that each of the threads of the process has executed in user mode is determined, and then all of those times are summed together to obtain this value.

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

All times are expressed using **FILETIME** data structures. Such a structure contains two 32-bit values that combine to

form a 64-bit count of 100-nanosecond time units.

Process creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. The Win32 API provides several functions that an application can use to convert such values to more generally useful forms.

Process kernel mode and user mode times are amounts of time. For example, if a process has spent one second in kernel mode, this function will fill the **FILETIME** structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, FILETIME, FileTimeToDosDateTime, FileTimeToLocalFileTime, FileTimeToSystemTime

---

1.174 GetProcessVersion

The **GetProcessVersion** function retrieves the major and minor version numbers of the system on which the specified process expects to run.

```
GetProcessVersion: procedure
(
    ProcessId:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessVersion@4" );
```

Parameters

ProcessId

[in] Process identifier that specifies the process of interest. A value of zero specifies the calling process.

Return Values

If the function succeeds, the return value is the version of the system on which the process expects to run. The high word of the return value contains the major version number. The low word of the return value contains the minor version number.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. The function fails if *ProcessId* is an invalid value.

Remarks

The **GetProcessVersion** function performs less quickly when *ProcessId* is nonzero, specifying a process other than the calling process.

The version number returned by this function is the version number stamped in the image header of the .exe file the process is running. Linker programs set this value.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions

1.175 GetProcessWorkingSetSize

The **GetProcessWorkingSetSize** function retrieves the minimum and maximum working set sizes of the specified process.

The "working set" of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The size of a process' working set is specified in bytes. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

```
GetProcessWorkingSetSize: procedure
(
        hProcess:                   dword;
    var lpMinimumWorkingSetSize:    SIZE_T;
    var lpMaximumWorkingSetSize:    SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProcessWorkingSetSize@12" );
```

Parameters

hProcess

[in] Handle to the process whose working set sizes will be obtained. The handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see Process Security and Access Rights.

lpMinimumWorkingSetSize

[out] Pointer to a variable that receives the minimum working set size of the specified process. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

lpMaximumWorkingSetSize

[out] Pointer to a variable that receives the maximum working set size of the specified process. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active when memory is in short supply.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, SetProcessWorkingSetSize

1.176 GetProfileInt

The **GetProfileInt** function retrieves an integer from a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetProfileInt: procedure
(
    lpAppName:  string;
    lpKeyName:  string;
    nDefault:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProfileIntA@12" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string that specifies the name of the section containing the key name.

lpKeyName

[in] Pointer to the null-terminated string specifying the name of the key whose value is to be retrieved. This value is in the form of a string; the **GetProfileInt** function converts the string into an integer and returns the integer.

nDefault

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

Return Values

The return value is the integer equivalent of the string following the key name in Win.ini. If the function cannot find the key, the return value is the default value. If the value of the key is less than zero, the return value is zero.

Remarks

If the key name consists of digits followed by characters that are not numeric, the function returns only the value of the digits. For example, the function returns 102 for the following line: *KeyName*=**102abc**.

Windows NT/2000: Calls to profile functions may be mapped to the registry instead of to the 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

When the operation has been mapped, the **GetProfileInt** 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileInt, WriteProfileString

---

1.177 GetProfileSection

The **GetProfileSection** function retrieves all the keys and values for the specified section of the Win.ini file.

Windows 95: The specified profile section must not exceed 32K.

Windows NT/2000: The specified profile section has no size limit.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetProfileSection: procedure
(
    lpAppName:          string;
    lpReturnedString:   string;
    nSize:              dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProfileSectionA@12" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string specifying the name of the section in the Win.ini file.

lpReturnedString

[out] Pointer to a buffer that receives the keys and values 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

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpReturnedString* parameter.

Windows 95: The maximum buffer size is 32,767 characters.

Return Values

The return value specifies the number of characters copied to the specified buffer, not including the terminating null character. If the buffer is not large enough to contain all the keys and values associated with the named section, the return value is equal to the length specified by *nSize* minus two.

Remarks

The format of the returned keys and values is one or more null-terminated strings, followed by a final null character. Each string has the following form:

key=string

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

This operation is atomic; no updates to the Win.ini file are allowed while the keys and values for the section are being copied to the buffer.

Windows NT/2000: Calls to profile functions may be mapped to the registry instead of to the 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

When the operation has been mapped, the **GetProfileSection** 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Registry Overview, Registry Functions, GetPrivateProfileSection, WriteProfileSection

---

1.178 GetProfileString

The **GetProfileString** function retrieves the string associated with a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Win32-based applications should store initialization information in the registry.

```
GetProfileString: procedure
(
        lpAppName:          string;
        lpKeyName:          string;
        lpDefault:          string;
    var lpReturnedString:   var;
        nSize:              dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetProfileStringA@20" );
```

Parameters

lpAppName

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

lpKeyName

[in] Pointer to a null-terminated string specifying the name of the key whose associated string is to be retrieved. If this parameter is NULL, the function copies all keys in the given section to the supplied buffer. Each string is followed by a null character, and the final string is followed by a second null character.

lpDefault

[in] Pointer to a null-terminated default string. If the *lpKeyName* key cannot be found in the initialization file, **GetProfileString** 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 *lpRe-*

turnedString buffer to strip any trailing blanks.

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

Windows NT/2000: The system 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

[out] Pointer to a buffer that receives the character string.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by the *lpReturnedString* parameter.

Return Values

The return value is the number of characters copied to the buffer, not including the null-terminating 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

If the string associated with the *lpKeyName* parameter is enclosed in single or double quotation marks, the marks are discarded when the **GetProfileString** function returns the string.

The **GetProfileString** function is not case-sensitive; the strings can contain a combination of uppercase and lowercase letters.

A section in the Win.ini file must have the following form:

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

An application can use the **GetPrivateProfileString** function to retrieve a string from a specified initialization file.

The *lpDefault* parameter must point to a valid string, even if the string is empty (that is, even if its first character is a null character).

Windows NT/2000: Calls to profile functions may be mapped to the registry instead of to the 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

When the operation has been mapped, the **GetProfileString** 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileString, WriteProfileString

---

1.179 GetQueuedCompletionStatus

The **GetQueuedCompletionStatus** function attempts to dequeue an I/O completion packet from a specified I/O completion port. If there is no completion packet queued, the function waits for a pending I/O operation associated with the completion port to complete.

```
GetQueuedCompletionStatus: procedure
(
        CompletionPort:     dword;
    var lpNumberOfBytes:    dword;
    var lpCompletionKey:    dword;
    var lpOverlapped:       OVERLAPPED;
        dwMilliseconds:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetQueuedCompletionStatus@20" );
```

Parameters

CompletionPort

[in] Handle to the completion port of interest. To create a completion port, use the **CreateIoCompletionPort** function.

lpNumberOfBytes

[out] Pointer to a variable that receives the number of bytes transferred during an I/O operation that has completed.

lpCompletionKey

[out] Pointer to a variable that receives the completion key value associated with the file handle whose I/O operation has completed. A completion key is a per-file key that is specified in a call to **CreateIoCompletionPort**.

lpOverlapped

[out] Pointer to a variable that receives the address of the **OVERLAPPED** structure that was specified when the completed I/O operation was started.

The following functions can be used to start I/O operations that complete using completion ports. You must pass the function an **OVERLAPPED** structure and a file handle associated with an completion port (by a call to **CreateIoCompletionPort**) to invoke the I/O completion port mechanism:

ConnectNamedPipe

DeviceIoControl

LockFileEx

ReadDirectoryChangesW

ReadFile

TransactNamedPipe

WaitCommEvent

WriteFile

Even if you have passed the function a file handle associated with a completion port and a valid **OVERLAPPED** structure, an application can prevent completion port notification. This is done by specifying a valid event handle for the **hEvent** member of the **OVERLAPPED** structure, and setting its low-order bit. A valid event handle whose low-order bit is set keeps I/O completion from being queued to the completion port.

dwMilliseconds

[in] Specifies the number of milliseconds that the caller is willing to wait for an completion packet to appear at the completion port. If a completion packet doesn't appear within the specified time, the function times out, returns FALSE, and sets **lpOverlapped* to NULL.

If *dwMilliseconds* is INFINITE, the function will never time out. If *dwMilliseconds* is zero and there is no I/O operation to dequeue, the function will time out immediately.

Return Values

If the function dequeues a completion packet for a successful I/O operation from the completion port, the return value is nonzero. The function stores information in the variables pointed to by the *lpNumberOfBytesTransferred*, *lpCompletionKey*, and *lpOverlapped* parameters.

If **lpOverlapped* is NULL and the function does not dequeue a completion packet from the completion port, the return value is zero. The function does not store information in the variables pointed to by the *lpNumberOfBytesTransferred* and *lpCompletionKey* parameters. To get extended error information, call **GetLastError**. If the function did not dequeue a completion packet because the wait timed out, **GetLastError** returns WAIT_TIMEOUT.

If **lpOverlapped* is not NULL and the function dequeues a completion packet for a failed I/O operation from the completion port, the return value is zero. The function stores information in the variables pointed to by *lpNumberOfBytesTransferred*, *lpCompletionKey*, and *lpOverlapped*. To get extended error information, call **GetLastError**.

Remarks

This function associates a thread with the specified completion port. A thread can be associated with at most one completion port.

The I/O system can be instructed to send completion notification packets to completion ports, where they are queued. The **CreateIoCompletionPort** function provides a mechanism for this.

When you perform an input/output operation with a file handle that has an associated input/output completion port, the I/O system sends a completion notification packet to the completion port when the I/O operation completes. The completion port places the completion packet in a first-in-first-out queue. The **GetQueuedCompletionStatus** function retrieves these queued completion packets.

A server application may have several threads calling **GetQueuedCompletionStatus** for the same completion port. As input operations complete, the operating system queues completion packets to the completion port. If threads are actively waiting in a call to this function, queued requests complete their call. For more information, see I/O Completion Ports.

You can call the **PostQueuedCompletionStatus** function to post an completion packet to an completion port. The completion packet will satisfy an outstanding call to the **GetQueuedCompletionStatus** function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, ConnectNamedPipe, CreateIoCompletionPort, DeviceIoControl, LockFileEx, OVERLAPPED, ReadFile, PostQueuedCompletionStatus, TransactNamedPipe, WaitCommEvent, WriteFile

---

1.180 GetShortPathName

The **GetShortPathName** function retrieves the short path form of a specified input path.

```
GetShortPathName: procedure
(
        lpszLongPath:   string;
    var lpszShortPath:  var;
        cchBuffer:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetShortPathNameA@12" );
```

Parameters

lpszLongPath

[in] Pointer to a null-terminated path string. The function retrieves the short form of this path.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpszShortPath

[out] Pointer to a buffer to receive the null-terminated short form of the path specified by *lpszLongPath*.

cchBuffer

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by *lpszShortPath*.

Return Values

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpszShortPath*, not including the terminating null character.

If the function fails due to the *lpszShortPath* buffer being too small to contain the short path string, the return value is the size, in **TCHARs**, of the short path string, including a terminating null. In this event, call the function again with a short path buffer that is at least as large as the return value times the size of a **TCHAR**.

If the function fails for any other reason, the return value is zero. To get extended error information, call **`GetLastError`**.

Remarks

When an application calls this function and specifies a path on a volume that does not support 8.3 aliases, the function fails with ERROR_INVALID_PARAMETER if the path is longer than 67 bytes.

The path specified by *lpszLongPath* does not have to be a full or a long path. The short form may be longer than the specifed path.

If the specified path is already in its short form, there is no need for any conversion, and the function simply copies the specified path to the buffer for the short path.

You can set *lpszShortPath* to the same value as *lpszLongPath*; in other words, you can set the buffer for the short path to the address of the input path string.

You can obtain the long name of a file from the short name by calling the **`GetLongPathName`** function. Alternatively, where **GetLongPathName** is not available, you can call **`FindFirstFile`** on each component of the path to get the corresponding long name.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetFullPathName, GetLongPathName, FindFirstFile

---

1.181 GetStartupInfo

The **GetStartupInfo** function retrieves the contents of the **`STARTUPINFO`** structure that was specified when the calling process was created.

```
GetStartupInfo: procedure
(
    var lpStartupInfo:  STARTUPINFO
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetStartupInfoA@4" );
```

Parameters

lpStartupInfo

[out] Pointer to a **STARTUPINFO** structure that receives the startup information.

Return Values

This function does not return a value.

Remarks

The **STARTUPINFO** structure was specified by the process that created the calling process. It can be used to specify properties associated with the main window of the calling process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, STARTUPINFO

1.182 GetStdHandle

The **GetStdHandle** function retrieves a handle for the standard input, standard output, or standard error device.

```
GetStdHandle: procedure
(
    nStdHandle:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetStdHandle@4" );
```

Parameters

nStdHandle

[in] Specifies the standard device for which to return the handle. This parameter can be one of the following values.

Value	Meaning
STD_INPUT_HANDLE	Standard input handle
STD_OUTPUT_HANDLE	Standard output handle
STD_ERROR_HANDLE	Standard error handle

Return Values

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

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

Remarks

Handles returned by **GetStdHandle** can be used by applications that need to read from or write to the console. When a console is created, the standard input handle is a handle to the console's input buffer, and the standard output and standard error handles are handles of the console's active screen buffer. These handles can be used by the **ReadFile** and **WriteFile** functions, or by any of the console functions that access the console input buffer or a screen buffer (for

example, the **ReadConsoleInput**, **WriteConsole**, or **GetConsoleScreenBufferInfo** functions).

All handles returned by this function have GENERIC_READ and GENERIC_WRITE access unless the **SetStdHandle** function has been used to set a standard handle to be some handle with a lesser access.

The standard handles of a process may be redirected by a call to **SetStdHandle**, in which case **GetStdHandle** returns the redirected handle. If the standard handles have been redirected, you can specify the CONIN$ value in a call to the **CreateFile** function to get a handle to a console's input buffer. Similarly, you can specify the CONOUT$ value to get a handle to a console's active screen buffer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CreateFile, GetConsoleScreenBufferInfo, ReadConsoleInput, ReadFile, SetStdHandle, WriteConsole, WriteFile

---

1.183 GetStringType

The **GetStringType** function retrieves character-type information for the characters in the specified source string. For each character in the string, the function sets one or more bits in the corresponding 16-bit element of the output array. Each bit identifies a given character type, such as whether the character is a letter, a digit, or neither.

```
GetStringType: procedure
(
        Locale:         LCID;
        dwInfoType:     dword;
        lpSrcStr:       string;
        cchSrc:         dword;
    var lpCharType:     var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetStringTypeA@20" );
```

Parameters

Locale

[in] Specifies the locale identifier. This value uniquely defines the ANSI code page to use to translate the string pointed to by *lpSrcStr* from ANSI to Unicode. The function then analyzes each Unicode character for character type information.

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale
LOCALE_USER_DEFAULT	Default user locale

Note that the *Locale* parameter does not exist in the **GetStringTypeW** function. Because of that parameter difference, an application cannot automatically invoke the proper **A** or **W** version of **GetStringType*** through the use of the **#define UNICODE** switch. An application can circumvent this limitation by using **GetStringTypeEx**, which is the

recommended function.

dwInfoType

[in] Specifies the type of character information the user wants to retrieve. The various types are divided into different levels (see the following Remarks section for a list of the information included in each type). This parameter can specify one of the following character type flags.

Flag	Meaning
CT_CTYPE1	Retrieve character type information.
CT_CTYPE2	Retrieve bidirectional layout information.
CT_CTYPE3	Retrieve text processing information.

lpSrcStr

[in] Pointer to the string for which character types are requested. If *cchSrc* is –1, the string is assumed to be null terminated. This must be an ANSI string. Note that this can be a double-byte character set (DBCS) string if the locale is appropriate for DBCS.

cchSrc

[in] Specifies the size, in characters, of the string pointed to by the *lpSrcStr* parameter. If this count includes a null terminator, the function returns character type information for the null terminator. If this value is –1, the string is assumed to be null terminated and the length is calculated automatically.

lpCharType

[out] Pointer to an array of 16-bit values. The length of this array must be large enough to receive one 16-bit value for each character in the source string. When the function returns, this array contains one word corresponding to each character in the source string.

Return Values

If the function succeeds, the return value is nonzero.

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

ERROR_INVALID_FLAGS
ERROR_INVALID_PARAMETER

Remarks

The *lpSrcStr* and *lpCharType* pointers must not be the same. If they are the same, the function fails and **GetLastError** returns ERROR_INVALID_PARAMETER.

The *Locale* parameter is only used to perform string conversion to Unicode. It has nothing to do with the CTYPEs the function returns. The CTYPEs are solely determined by Unicode code points, and do not vary on a locale basis. For example, Greek letters are C1_ALPHA for any *Locale* value.

The character-type bits are divided into several levels. The information for one level can be retrieved by a single call to this function. Each level is limited to 16 bits of information so that the other mapping routines, which are limited to 16 bits of representation per character, can also return character-type information.

The character types supported by this function include the following.

Ctype 1

These types support ANSI C and POSIX (LC_CTYPE) character-typing functions. A bitwise-OR of these values is returned in the array pointed to by the *lpCharType* parameter when the *dwInfoType* parameter is set to CT_CTYPE1. For DBCS locales, the Ctype 1 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters all have the C1_ALPHA attribute.

Name	Value	Meaning
C1_UPPER	0x0001	Uppercase
C1_LOWER	0x0002	Lowercase
C1_DIGIT	0x0004	Decimal digits
C1_SPACE	0x0008	Space characters
C1_PUNCT	0x0010	Punctuation
C1_CNTRL	0x0020	Control characters
C1_BLANK	0x0040	Blank characters
C1_XDIGIT	0x0080	Hexadecimal digits
C1_ALPHA	0x0100	Any linguistic character: alphabetic, syllabary, or ideographic

The following character types are either constant or computable from basic types and do not need to be supported by this function.

Type	Description
Alphanumeric	Alphabetic characters and digits (C1_ALPHA and C1_DIGIT).
Printable	Graphic characters and blanks (all C1_* types except C1_CNTRL).

The Windows version 3.1 functions **IsCharUpper** and **IsCharLower** do not always produce correct results for characters in the range 0x80-0x9f, so they may produce different results than this function for characters in that range. (For example, the German Windows version 3.1 language driver incorrectly reports 0x9a, lowercase s hacek, as uppercase).

Ctype 2

These types support proper layout of Unicode text. For DBCS locales, Ctype 2 applies to both narrow and wide characters. The direction attributes are assigned so that the bidirectional layout algorithm standardized by Unicode produces accurate results. These types are mutually exclusive. For more information about the use of these attributes, see *The Unicode Standard: Worldwide Character Encoding, Volumes 1 and 2,* Addison Wesley Publishing Company: 1991, 1992, ISBN 0201567881.

Name	Value	Meaning
Strong		
C2_LEFTTORIGHT	0x0001	Left to right
C2_RIGHTTOLEFT	0x0002	Right to left
Weak		
C2_EUROPENUMBER	0x0003	European number, European digit
C2_EUROPESEPARATOR	0x0004	European numeric separator
C2_EUROPETERMINATOR	0x0005	European numeric terminator
C2_ARABICNUMBER	0x0006	Arabic number
C2_COMMONSEPARATOR	0x0007	Common numeric separator

Neutral		
C2_BLOCKSEPARATOR	0x0008	Block separator
C2_SEGMENTSEPARATOR	0x0009	Segment separator
C2_WHITESPACE	0x000A	White space
C2_OTHERNEUTRAL	0x000B	Other neutrals
Not applicable		
C2_NOTAPPLICABLE	0x0000	No implicit directionality (for example, control codes)

Ctype 3

These types are intended to be placeholders for extensions to the POSIX types required for general text processing or for the standard C library functions. A bitwise-OR of these values is returned when *dwInfoType* is set to CT_CTYPE3. For DBCS locales, the Cypte 3 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters all have the C3_ALPHA attribute.

Name	Value	Meaning
C3_NONSPACING	0x0001	Nonspacing mark.
C3_DIACRITIC	0x0002	Diacritic nonspacing mark.
C3_VOWELMARK	0x0004	Vowel nonspacing mark.
C3_SYMBOL	0x0008	Symbol.
C3_KATAKANA	0x0010	Katakana character.
C3_HIRAGANA	0x0020	Hiragana character.
C3_HALFWIDTH	0x0040	Half-width (narrow) character.
C3_FULLWIDTH	0x0080	Full-width (wide) character.
C3_IDEOGRAPH	0x0100	Ideographic character.
C3_KASHIDA	0x0200	Arabic Kashida character.
C3_LEXICAL	0x0400	Punctuation which is counted as part of the word (Kashida, hyphen, feminine/masculine ordinal indicators, equal sign, and so forth).
C3_ALPHA	0x8000	All linguistic characters (alphabetic, syllabary, and ideographic).
Not applicable		
C3_NOTAPPLICABLE	0x0000	Not applicable.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, GetLocaleInfo, GetStringTypeEx, GetStringTypeW

1.184 GetSystemDefaultLCID

The **GetSystemDefaultLCID** function retrieves the system default locale identifier.

```
GetSystemDefaultLCID: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemDefaultLCID@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the system default locale identifier. If the function fails, the return value is zero.

Remarks

For more information about locale identifiers, see Locales.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winnls.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, ConvertDefaultLocale, GetLocaleInfo, GetUserDefaultLCID, MAKELCID

1.185 GetSystemDefaultLangID

The **GetSystemDefaultLangID** function retrieves the language identifier of the system locale.

```
GetSystemDefaultLangID: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemDefaultLangID@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the language identifier of the system locale. If the function fails, the return value is zero.

Remarks

For more information about language identifiers, see Language Identifiers and Locale Information.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetSystemDefaultLCID, GetUserDefaultLangID, MAKELANGID

1.186 GetSystemDirectory

The **GetSystemDirectory** function retrieves the path of the system directory. The system directory contains such files as dynamic-link libraries, drivers, and font files.

```
GetSystemDirectory: procedure
(
    var lpBuffer:   var;
        uSize:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemDirectoryA@8" );
```

Parameters

lpBuffer

[out] Pointer to the buffer to receive the null-terminated string containing the path. This path does not end with a backslash unless the system directory is the root directory. For example, if the system directory is named WINDOWS\SYSTEM on drive C, the path of the system directory retrieved by this function is C:\WINDOWS\SYSTEM.

uSize

[in] Specifies the maximum size of the buffer, in **TCHARs**. This value should be set to at least MAX_PATH.

Return Values

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to the buffer, not including the terminating null character. If the length is greater than the size of the buffer, the return value is the size of the buffer required to hold the path.

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

Remarks

Applications should *not* create files in the system directory. If the user is running a shared version of the operating system, the application does not have write access to the system directory. Applications should create files only in the directory returned by the **GetWindowsDirectory** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetCurrentDirectory, GetWindowsDirectory, SetCurrentDirectory

---

1.187 GetSystemInfo

The **GetSystemInfo** function returns information about the current system.

```
GetSystemInfo: procedure
(
    var lpSystemInfo:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemInfo@4" );
```

Parameters

lpSystemInfo

[out] Pointer to a **SYSTEM_INFO** structure that receives the information.

Return Values

This function does not return a value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, SYSTEM_INFO

1.188 GetSystemPowerStatus

The **GetSystemPowerStatus** function retrieves the power status of the system. The status indicates whether the system is running on AC or DC power, whether the battery is currently charging, and how much battery life remains.

```
GetSystemPowerStatus: procedure
(
    var lpSystemPowerStatus:    SYSTEM_POWER_STATUS
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemPowerStatus@4" );
```

Parameters

lpSystemPowerStatus

[out] Pointer to a **SYSTEM_POWER_STATUS** structure that receives status information.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Power Management Overview, Power Management Functions, SYSTEM_POWER_STATUS

1.189 GetSystemTime

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

```
GetSystemTime: procedure
(
    var lpSystemTime:   SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemTime@4" );
```

Parameters

lpSystemTime

[out] Pointer to a **SYSTEMTIME** structure to receive the current system date and time.

Return Values

This function does not return a value.

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetLocalTime, GetSystemTimeAdjustment, SetSystemTime, SYSTEMTIME

1.190 GetSystemTimeAdjustment

The **GetSystemTimeAdjustment** function determines whether the system is applying periodic time adjustments to its time-of-day clock at each clock interrupt, along with the value and period of any such adjustments. Note that the

period of such adjustments is equivalent to the time period between clock interrupts.

```
GetSystemTimeAdjustment: procedure
(
    var lpTimeAdjustment:           dword;
    var lpTimeIncrement:            dword;
    var lpTimeAdjustmentDisabled:   boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemTimeAdjustment@12" );
```

Parameters

lpTimeAdjustment

[out] Pointer to a **DWORD** that the function sets to the number of 100-nanosecond units added to the time-of-day clock at each periodic time adjustment.

lpTimeIncrement

[out] Pointer to a **DWORD** that the function sets to the interval, counted in 100-nanosecond units, between periodic time adjustments. This interval is the time period between a system's clock interrupts.

lpTimeAdjustmentDisabled

[out] Pointer to a **BOOL** that the function sets to indicate whether periodic time adjustment is in effect.

A value of TRUE indicates that periodic time adjustment is disabled. At each clock interrupt, the system merely adds the interval between clock interrupts to the time-of-day clock. The system is free, however, to adjust its time-of-day clock using other techniques. Such other techniques may cause the time-of-day clock to noticeably jump when adjustments are made.

A value of FALSE indicates that periodic time adjustment is being used to adjust the time-of-day clock. At each clock interrupt, the system adds the time increment specified by **SetSystemTimeAdjustment**'s *dwTimeIncrement* parameter to the time-of-day clock. The system will not interfere with the time adjustment scheme, and will not attempt to synchronize time of day on its own via other techniques.

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 **GetSystemTimeAdjustment** and **SetSystemTimeAdjustment** functions support algorithms that want to synchronize the time-of-day clock, reported by **GetSystemTime** and **GetLocalTime**, with another time source using a periodic time adjustment applied at each clock interrupt.

When periodic time adjustment is in effect, the system adds an adjusting value to the time-of-day clock at a periodic interval, at each clock interrupt. The **GetSystemTimeAdjustment** function lets a caller determine whether periodic time adjustment is enabled, and if it is, obtain the amount of each adjustment and the time between adjustments. The **SetSystemTimeAdjustment** function lets a caller enable or disable periodic time adjustment, and set the value of the adjusting increment.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, SetSystemTimeAdjustment, GetSystemTime, GetLocalTime

1.191 GetSystemTimeAsFileTime

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

```
GetSystemTimeAsFileTime: procedure
(
    var lpSystemTimeAsFileTime: FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetSystemTimeAsFileTime@4" );
```

Parameters

lpSystemTimeAsFileTime

[out] 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.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, GetSystemTime, SYSTEMTIME, SystemTimeToFileTime

1.192 GetTapeParameters

The **GetTapeParameters** function retrieves information that describes the tape or the tape drive.

```
GetTapeParameters: procedure
(
        hDevice:            dword;
        dwOperation:        dword;
    var lpdwSize:           dword;
    var lpTapeInformation:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTapeParameters@16" );
```

Parameters

hDevice

[in] Handle to the device about which information is sought. This handle is created by using the **CreateFile** function.

dwOperation

[in] Specifies the type of information requested. This parameter must be one of the following values.

Value	Description
GET_TAPE_MEDIA_INFORMATION	Retrieves information about the tape in the tape device.
GET_TAPE_DRIVE_INFORMATION	Retrieves information about the tape device.

lpdwSize

[out] Pointer to a variable that receives the size, in bytes, of the buffer specified by the *lpTapeInformation* parameter. If the buffer is too small, this parameter receives the required size.

lpTapeInformation

[out] Pointer to a structure that contains the requested information. If the *dwOperation* parameter is GET_TAPE_MEDIA_INFORMATION, *lpTapeInformation* points to a **TAPE_GET_MEDIA_PARAMETERS** structure.

If *dwOperation* is GET_TAPE_DRIVE_INFORMATION, *lpTapeInformation* points to a **TAPE_GET_DRIVE_PARAMETERS** structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error code	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.

ERROR_WRITE_PROTECT	The media is write protected.

Remarks

The block size range values (maximum and minimum) returned by the **GetTapeParameters** function called with the *dwOperation* parameter set to the GET_TAPE_DRIVE_INFORMATION value will indicate system limits, not drive limits. However, it is the tape drive device and the media present in the drive that determine the true block size limits. Thus, an application may not be able to set all the block sizes mentioned in the range obtained by specifying GET_TAPE_DRIVE_INFORMATION in *dwOperation.*

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, SetTapeParameters, TAPE_GET_DRIVE_PARAMETERS, TAPE_GET_MEDIA_PARAMETERS

1.193 GetTapePosition

The **GetTapePosition** function retrieves the current address of the tape, in logical or absolute blocks.

```
GetTapePosition: procedure
(
        hDevice:        dword;
        dwPositionType: dword;
    var lpdwPartition:  dword;
    var lpdwOffsetLow:  dword;
    var lpdwOffsetHigh: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTapePosition@20" );
```

Parameters

hDevice

[in] Handle to the device on which to get the tape position. This handle is created by using CreateFile.

dwPositionType

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

Value	Description
TAPE_ABSOLUTE_POSITION	The *lpdwOffsetLow* and *lpdwOffsetHigh* parameters receive the device-specific block address. The *dwPartition* parameter receives zero.
TAPE_LOGICAL_POSITION	The *lpdwOffsetLow* and *lpdwOffsetHigh* parameters receive the logical block address. The *dwPartition* parameter receives the logical tape partition.

lpdwPartition

[out] Pointer to a variable that receives the number of the current tape partition. Partitions are numbered logically

from 1 through *n*, where 1 is the first partition on the tape and *n* is the last. When a device-specific block address is retrieved, or if the device supports only one partition, this parameter receives zero.

lpdwOffsetLow

[out] Pointer to a variable that receives the low-order bits of the current tape position.

lpdwOffsetHigh

[out] Pointer to a variable that receives the high-order bits of the current tape position. This parameter can be NULL if the high-order bits are not required.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error code	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

A logical block address is relative to a partition. The first logical block address on each partition is zero.

Call the **GetTapeParameters** function to obtain information about the status, capabilities, and capacities of tape drives and media.

Requirements

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

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, GetTapeParameters, SetTapePosition

---

1.194 GetTapeStatus

The **GetTapeStatus** function determines whether the tape device is ready to process tape commands.

```
GetTapeStatus: procedure
(
    hDevice:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTapeStatus@4" );
```

Parameters

hDevice

[in] Handle to the device for which to get the device status. This handle is created by using the **`CreateFile`** function.

Return Values

If the tape device is ready to accept appropriate tape-access commands without returning errors, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error code	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_DEVICE_REQUIRES_CLEANING	The tape drive is capable of reporting that it requires cleaning, and reports that it does require cleaning.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.

ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile

1.195 GetTempFileName

The **GetTempFileName** function creates a name for a temporary file. The file name is the concatenation of specified path and prefix strings, a hexadecimal string formed from a specified integer, and the .tmp extension.

The specified integer can be nonzero, in which case, the function creates the file name but does not create the file. If you specify zero for the integer, the function creates a unique file name and creates the file in the specified directory.

```
GetTempFileName: procedure
(
        lpPathName:     string;
        lpPrefixString: string;
        uUnique:        dword;
    var lpTempFileName: var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTempFileNameA@16" );
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the directory path for the file name. This string must consist of characters in the ANSI character set. Applications typically specify a period (.) or the result of the **GetTempPath** function for this parameter. If this parameter is NULL, the function fails.

lpPrefixString

[in] Pointer to a null-terminated prefix string. The function uses the first three characters of this string as the prefix of the file name. This string must consist of characters in the ANSI character set.

uUnique

[in] Specifies an unsigned integer that the function converts to a hexadecimal string for use in creating the temporary file name.

If *uUnique* is nonzero, the function appends the hexadecimal string to *lpPrefixString* to form the temporary file name. In this case, the function does not create the specified file, and does not test whether the file name is unique.

If *uUnique* is zero, the function uses a hexadecimal string derived from the current system time. In this case, the function uses different values until it finds a unique file name, and then it creates the file in the *lpPathName* directory.

lpTempFileName

[out] Pointer to the buffer that receives the temporary file name. This null-terminated string consists of characters in the ANSI character set. This buffer should be at least the length, in bytes, specified by MAX_PATH to accommodate the path.

Return Values

If the function succeeds, the return value specifies the unique numeric value used in the temporary file name. If the *uUnique* parameter is nonzero, the return value specifies that same number.

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

Remarks

The **GetTempFileName** function creates a temporary file name of the following form:

```
path\preuuuu.TMP
```

The following table describes the file name syntax.

Component	Meaning
path	Path specified by the *lpPathName* parameter
pre	First three letters of the *lpPrefixString* string
uuuu	Hexadecimal value of *uUnique*

When the system shuts down, temporary files whose names have been created by this function are not automatically deleted.

To avoid problems resulting when converting an ANSI string, an application should call the **CreateFile** function to create a temporary file.

If the *uUnique* parameter is zero, **GetTempFileName** attempts to form a unique number based on the current system time. If a file with the resulting file name exists, the number is increased by one and the test for existence is repeated. Testing continues until a unique file name is found. **GetTempFileName** then creates a file by that name and closes it. When *uUnique* is nonzero, no attempt is made to create and open the file.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, GetTempPath

1.196 GetTempPath

The **GetTempPath** function retrieves the path of the directory designated for temporary files.

```
GetTempPath: procedure
(
        nBufferLength:  dword;
    var lpBuffer:       dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTempPathA@8" );
```

Parameters

nBufferLength

[in] Specifies the size, in **TCHARs**, of the string buffer identified by *lpBuffer*.

lpBuffer

[out] Pointer to a string buffer that receives the null-terminated string specifying the temporary file path. The returned string ends with a backslash, for example, C:\TEMP\.

Return Values

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nBufferLength*, the return value is the size of the buffer required to hold the path.

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

Remarks

Windows 95/98: The **GetTempPath** function gets the temporary file path as follows:

The path specified by the TMP environment variable.

The path specified by the TEMP environment variable, if TMP is not defined or if TMP specifies a directory that does not exist.

The current directory, if both TMP and TEMP are not defined or specify nonexistent directories.

Windows NT/2000: The **GetTempPath** function does not verify that the directory specified by the TMP or TEMP environment variables exists. The function gets the temporary file path as follows:

The path specified by the TMP environment variable.

The path specified by the TEMP environment variable, if TMP is not defined.

The Windows directory, if both TMP and TEMP are not defined.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetTempFileName

1.197 GetThreadContext

The **GetThreadContext** function retrieves the context of the specified thread.

```
GetThreadContext: procedure
(
        hThread:    dword;
    var lpContext:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadContext@8" );
```

Parameters

hThread

[in] Handle to the thread whose context is to be retrieved.

Windows NT/ 2000: The handle must have THREAD_GET_CONTEXT access to the thread. For more information, see Thread Security and Access Rights.

lpContext

[in/out] Pointer to the **CONTEXT** structure that receives the appropriate context of the specified thread. The value of the **ContextFlags** member of this structure specifies which portions of a thread's context are retrieved. The **CONTEXT** structure is highly computer specific. Currently, there are **CONTEXT** structures defined for Intel, MIPS, Alpha, and PowerPC processors. Refer to the WinNt.h header file for definitions of these structures.

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 **GetThreadContext** function is used to retrieve the context of the specified thread. The function allows a selective context to be retrieved based on the value of the **ContextFlags** member of the **CONTEXT** structure. The thread handle identified by the *hThread* parameter is typically being debugged, but the function can also operate when it is not being debugged.

You cannot get a valid context for a running thread. Use the **SuspendThread** function to suspend the thread before calling **GetThreadContext**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, CONTEXT, SetThreadContext, SuspendThread

1.198 GetThreadLocale

The **GetThreadLocale** function retrieves the calling thread's current locale.

```
GetThreadLocale: procedure;
```

```
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadLocale@0" );
```

Parameters

This function has no parameters.

Return Values

The function returns the calling thread's locale identifier.

Remarks

When a thread is created, it uses the system default–thread locale. The system reads the system default–thread locale from the registry when the system boots. This system default can be modified for future process and thread creation using Control Panel's International application.

For more information about locale identifiers, see Locales.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, SetThreadLocale, GetSystemDefaultLCID, GetUserDefaultLCID

1.199 GetThreadPriority

The **GetThreadPriority** function retrieves the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base-priority level.

```
GetThreadPriority: procedure
(
    hThread:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadPriority@4" );
```

Parameters

hThread

[in] Handle to the thread.

Windows NT/2000: The handle must have THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

Return Values

If the function succeeds, the return value is the thread's priority level.

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

The thread's priority level is one of the following values:

Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.
THREAD_PRIORITY_IDLE	Indicates a base-priority level of 1 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base-priority level of 16 for REALTIME_PRIORITY_CLASS processes.
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.
THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base-priority level of 15 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base-priority level of 31 for REALTIME_PRIORITY_CLASS processes.

Remarks

Every thread has a base-priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base-priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base-priority levels for each combination of priority class and thread priority value, refer to the **`SetPriorityClass`** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetPriorityClass, OpenThread, SetPriorityClass, SetThreadPriority

1.200 GetThreadPriorityBoost

The **GetThreadPriorityBoost** function retrieves the priority boost control state of the specified thread.

```
GetThreadPriorityBoost: procedure
(
        hThread:                dword;
    var pDisablePriorityBoost:  boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadPriorityBoost@8" );
```

Parameters

hThread

[in] Handle to the thread. This thread must have THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

pDisablePriorityBoost

[out] Pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return Values

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, OpenThread, SetThreadPriorityBoost

1.201 GetThreadSelectorEntry

The **GetThreadSelectorEntry** function retrieves a descriptor table entry for the specified selector and thread.

```
GetThreadSelectorEntry: procedure
(
        hThread:            dword;
        dwSelector:         dword;
    var lpSelectorEntry:    LDT_ENTRY
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadSelectorEntry@12" );
```

Parameters

hThread

[in] Handle to the thread containing the specified selector.

Windows NT/ 2000: The handle must have THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

dwSelector

[in] Specifies the global or local selector value to look up in the thread's descriptor tables.

lpSelectorEntry

[out] Pointer to an LDT_ENTRY structure that receives a copy of the descriptor table entry if the specified selector has an entry in the specified thread's descriptor table. This information can be used to convert a segment-relative address to a linear virtual address.

Return Values

If the function succeeds, the return value is nonzero. In that case, the structure pointed to by the *lpSelectorEntry* parameter receives a copy of the specified descriptor table entry.

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

Remarks

GetThreadSelectorEntry is only functional on x86-based systems. For systems that are not x86-based, the function returns FALSE.

Debuggers use this function to convert segment-relative addresses to linear virtual addresses. The **ReadProcessMemory** and **WriteProcessMemory** functions use linear virtual addresses.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, LDT_ENTRY, ReadProcessMemory, WriteProcessMemory

---

1.202 GetThreadTimes

The **GetThreadTimes** function retrieves timing information for the specified thread.

```
GetThreadTimes: procedure
(
        hThread:        dword;
    var lpCreationTime: FILETIME;
    var lpExitTime:     FILETIME;
    var lpKernelTime:   FILETIME;
    var lpUserTime:     FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetThreadTimes@20" );
```

Parameters

hThread

[in] Handle to the thread whose timing information is sought. This handle must be created with

THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

lpCreationTime

[out] Pointer to a **FILETIME** structure that receives the creation time of the thread.

lpExitTime

[out] Pointer to a **FILETIME** structure that receives the exit time of the thread. If the thread has not exited, the content of this structure is undefined.

lpKernelTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the thread has executed in kernel mode.

lpUserTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the thread has executed in user mode.

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

All times are expressed using **FILETIME** data structures. Such a structure contains two 32-bit values that combine to form a 64-bit count of 100-nanosecond time units.

Thread creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. The Win32 API provides several functions that an application can use to convert such values to more generally useful forms; see Time Functions, particularly those noted in the following See Also section.

Thread kernel mode and user mode times are amounts of time. For example, if a thread has spent one second in kernel mode, this function will fill the **FILETIME** structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, FILETIME, FileTimeToDosDateTime, FileTimeToLocalFileTime, FileTimeToSystemTime, OpenThread

---

1.203 GetTickCount

The **GetTickCount** function retrieves the number of milliseconds that have elapsed since the system was started. It is limited to the resolution of the system timer. To obtain the system timer resolution, use the **GetSystemTimeAdjustment** function.

```
GetTickCount: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetTickCount@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the number of milliseconds that have elapsed since the system was started.

Remarks

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

If you need a higher resolution timer, use a multimedia timer or a high-resolution timer.

Windows NT/2000: To obtain the time elapsed since the computer was started, retrieve the System Up Time counter in the performance data in the registry key **HKEY_PERFORMANCE_DATA**. The value returned is an 8-byte value. For more information, see Performance Monitoring.

Example

The following example demonstrates how to handle timer wrap around.

```
DWORD dwStart = GetTickCount();


// Stop if this has taken too long
if( GetTickCount() - dwStart >= TIMELIMIT )
    Cancel();
```

Where TIMELIMIT is the time interval of interest to the application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions

1.204 GetTimeFormat

The **GetTimeFormat** function formats time as a time string for a specified locale. The function formats either a specified time or the local system time.

```
GetTimeFormat: procedure
(
        Locale:         LCID;
        dwFlags:        dword;
    var lpTime:         SYSTEMTIME;
        lpFormat:       string;
    var lpTimeStr:      var;
        cchTime:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTimeFormatA@24" );
```

Parameters

Locale

[in] Specifies the locale for which the time string is to be formatted. If *lpFormat* is NULL, the function formats the string according to the time format for this locale. If *lpFormat* is not NULL, the function uses the locale only for information not specified in the format picture string (for example, the locale's time markers).

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

dwFlags

[in] Specifies various function options. You can specify a combination of the following values.

Value	Meaning
LOCALE_NOUSEROVERRIDE	If set, the function formats the string using the system default time format for the specified locale. If not set, the function formats the string using any user overrides to the locale's default time format. This flag cannot be set if *lpFormat* is non-NULL.
LOCALE_USE_CP_ACP	Uses the system ANSI code page for string translation instead of the locale code page.
TIME_NOMINUTESORSECONDS	Does not use minutes or seconds.
TIME_NOSECONDS	Does not use seconds.
TIME_NOTIMEMARKER	Does not use a time marker.
TIME_FORCE24HOURFORMAT	Always uses a 24-hour time format.

lpTime

[in] Pointer to a **SYSTEMTIME** structure that contains the time information to be formatted. If this pointer is NULL, the function uses the current local system time.

lpFormat

[in] Pointer to a format picture to use to form the time string. If *lpFormat* is NULL, the function uses the time format of the specified locale.

Use the following elements to construct a format picture string. If you use spaces to separate the elements in the format string, these spaces will appear in the same location in the output string. The letters must be in uppercase or lowercase as shown (for example, "ss", not "SS"). Characters in the format string that are enclosed in single quotation marks will appear in the same location and unchanged in the output string.

Picture	Meaning
h	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	Hours with leading zero for single-digit hours; 12-hour clock.
H	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	Hours with leading zero for single-digit hours; 24-hour clock.

m	Minutes with no leading zero for single-digit minutes.
mm	Minutes with leading zero for single-digit minutes.
s	Seconds with no leading zero for single-digit seconds.
ss	Seconds with leading zero for single-digit seconds.
t	One character time-marker string, such as A or P.
tt	Multicharacter time-marker string, such as AM or PM.

For example, to get the time string

```
"11:29:40 PM"
```

use the following picture string:

```
"hh':'mm':'ss tt"
```

lpTimeStr

[out] Pointer to a buffer that receives the formatted time string.

cchTime

[in] Specifies the size, in **TCHARs**, of the *lpTimeStr* buffer. If *cchTime* is zero, the function returns the number of bytes or characters required to hold the formatted time string, and the buffer pointed to by *lpTimeStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the buffer pointed to by *lpTimeStr*. If the *cchTime* parameter is zero, the return value is the number of bytes or characters required to hold the formatted time string. The count includes the terminating null.

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

Remarks

If a time marker exists and the TIME_NOTIMEMARKER flag is not set, the function localizes the time marker based on the specified locale identifier. Examples of time markers are "AM" and "PM" for US English.

The time values in the **SYSTEMTIME** structure pointed to by *lpTime* must be valid. The function checks each of the time values to determine that it is within the appropriate range of values. If any of the time values are outside the correct range, the function fails, and sets the last-error to ERROR_INVALID_PARAMETER.

The function ignores the date portions of the **SYSTEMTIME** structure pointed to by *lpTime*: **wYear**, **wMonth**, **wDayOfWeek**, and **wDay**.

If TIME_NOMINUTESORSECONDS or TIME_NOSECONDS is specified, the function removes the separator(s) preceding the minutes and/or seconds element(s).

If TIME_NOTIMEMARKER is specified, the function removes the separator(s) preceding and following the time marker.

If TIME_FORCE24HOURFORMAT is specified, the function displays any existing time marker, unless the TIME_NOTIMEMARKER flag is also set.

The function does not include milliseconds as part of the formatted time string.

To use LOCALE_NOUSEROVERRIDE, *lpFormat* must be NULL.

No errors are returned for a bad format string. The function simply forms the best time string that it can. If more than two hour, minute, second, or time marker format pictures are passed in, then the function defaults to two. For exam-

ple, the only time marker pictures that are valid are L"t" and L"tt" (the 'L' indicates a Unicode (16-bit characters) string). If L"ttt" is passed in, the function assumes L"tt".

To obtain the time format without performing any actual formatting, use the **GetLocaleInfo** function with the LOCALE_STIMEFORMAT parameter.

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only locale. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

National Language Support Overview, National Language Support Functions, GetDateFormat, GetLocaleInfo, SYSTEMTIME

1.205 GetTimeZoneInformation

The **GetTimeZoneInformation** function retrieves the current time-zone parameters. These parameters control the translations between Coordinated Universal Time (UTC) and local time.

```
GetTimeZoneInformation: procedure
(
    var lpTimeZoneInformation:  TIME_ZONE_INFORMATION
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetTimeZoneInformation@4" );
```

Parameters

lpTimeZoneInformation

[out] Pointer to a **TIME_ZONE_INFORMATION** structure to receive the current time-zone parameters.

Return Values

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

Value	Meaning
TIME_ZONE_ID_UNKNOWN	The system cannot determine the current time zone. This error is also returned if you call the **SetTimeZoneInformation** function and supply the bias values but no transition dates. **Windows NT/2000:** This value is returned if daylight saving time is not used in the current time zone, because there are no transition dates.

TIME_ZONE_ID_STANDARD	The system is operating in the range covered by the **StandardDate** member of the **TIME_ZONE_INFORMATION** structure. **Windows 95:** This value is returned if daylight saving time is not used in the current time zone, because there are no transition dates.
TIME_ZONE_ID_DAYLIGHT	The system is operating in the range covered by the **DaylightDate** member of the **TIME_ZONE_INFORMATION** structure.

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

Remarks

All translations between UTC time and local time are based on the following formula:

```
UTC = local time + bias
```

The bias is the difference, in minutes, between UTC time and local time.

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, SetTimeZoneInformation, TIME_ZONE_INFORMATION

1.206 GetUserDefaultLCID

The **GetUserDefaultLCID** function retrieves the user default–locale identifier.

```
GetUserDefaultLCID: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetUserDefaultLCID@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the user default–locale identifier. If the function fails, the return value is zero.

Remarks

On single-user systems, the return value is the same as that returned by **GetSystemDefaultLCID**.

For more information about locale identifiers, see Locales.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, ConvertDefaultLocale, GetLocaleInfo, GetSystemDefaultLCID, MAKELCID

1.207 GetUserDefaultLangID

The **GetUserDefaultLangID** function retrieves the language identifier of the current user locale.

```
GetUserDefaultLangID: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetUserDefaultLangID@0" );
```

Parameters

This function has no parameters.

Return Values

If the function is successful, the return value is the language identifier of the current user locale. If the function fails, the return value is zero.

Remarks

The return value is not necessarily the same as that returned by **GetSystemDefaultLangID**, even if the computer is a single-user system.

For more information about language identifiers, see Language Identifiers and Locale Information.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetSystemDefaultLangID, MAKE-LANGID

1.208 GetVersion

The **GetVersion** function returns the current version number of the operating system.

Note This function has been superseded by **GetVersionEx**. New applications should use **GetVersionEx** or **VerifyVersionInfo**.

```
GetVersion: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__GetVersion@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a **DWORD** value that contains the major and minor version numbers of the operating system in the low order word, and information about the operating system platform in the high order word.

For all platforms, the low order word contains the version number of the operating system. The low-order byte of this word specifies the major version number, in hexadecimal notation. The high-order byte specifies the minor version (revision) number, in hexadecimal notation.

To distinguish between operating system platforms, use the high order bit and the low order byte, as shown in the following table:

Platform	High-order bit	Low-order byte (major version)
Windows NT/2000	0	3, 4, or 5
Windows 95/98	1	4
Win32s with Windows 3.1	1	3

Windows NT/2000: The remaining bits in the high-order word specify the build number.

Windows 95/98: The remaining bits of the high-order word are reserved.

Remarks

The **`GetVersionEx`** function was developed because many existing applications err when examining the packed **DWORD** value returned by **GetVersion**, transposing the major and minor version numbers. **GetVersionEx** forces applications to explicitly examine each element of version information. **`VerifyVersionInfo`** eliminates further potential for error by comparing the required system version with the current system version for you.

The following code fragment illustrates how to extract information from the **GetVersion** return value:

```
dwVersion = GetVersion();


// Get the Windows version.

dwWindowsMajorVersion =  (DWORD)(LOBYTE(LOWORD(dwVersion)));
dwWindowsMinorVersion =  (DWORD)(HIBYTE(LOWORD(dwVersion)));

// Get the build number for Windows NT/Windows 2000 or Win32s.

if (dwVersion < 0x80000000)                // Windows NT/2000
    dwBuild = (DWORD)(HIWORD(dwVersion));
else if (dwWindowsMajorVersion < 4)        // Win32s
    dwBuild = (DWORD)(HIWORD(dwVersion) & ~0x8000);
else                  // Windows 95/98 -- No build number
    dwBuild =  0;
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetVersionEx, VerifyVersionInfo

---

1.209 GetVersionEx

The **GetVersionEx** function obtains extended information about the version of the operating system that is currently running.

Windows 2000: To compare the current system version to a required version, use the **VerifyVersionInfo** function instead of using **GetVersionEx** to perform the comparison yourself.

```
GetVersionEx: procedure
(
    var lpVersionInfo:  OSVERSIONINFO
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetVersionExA@4" );
```

Parameters

lpVersionInfo

[in/out] Pointer to an **OSVERSIONINFO** data structure that the function fills with operating system version information.

Before calling the **GetVersionEx** function, set the **dwOSVersionInfoSize** member of the **OSVERSIONINFO** data structure to `sizeof(OSVERSIONINFO)`.

Windows NT 4.0 SP6 and Windows 2000: This member can be a pointer to an **OSVERSIONINFOEX** structure. Set the **dwOSVersionInfoSize** member to `sizeof(OSVERSIONINFOEX)` to identify the structure type.

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**. The function fails if you specify an invalid value for the **dwOSVersionInfoSize** member of the **OSVERSIONINFO** or **OSVERSIONINFOEX** structure.

Remarks

When using the **GetVersionEx** function to determine whether your application is running on a particular version of the operating system, check for version numbers that are greater than or equal to the desired version numbers. This ensures that the test succeeds for later versions of the operating system. For example, if your application requires Windows 98, use the following test:

```
osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO)

GetVersionEx (&osvi);

bIsWindows98orLater =

   (osvi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) &&
```

```
( (osvi.dwMajorVersion > 4) ||
( (osvi.dwMajorVersion == 4) && (osvi.dwMinorVersion > 0) ) );
```

Identifying the current operating system is usually not the best way to determine whether a particular operating system feature is present. This is because the operating system may have had new features added in a redistributable DLL. Rather than using **GetVersionEx** to determine the operating system platform or version number, test for the presence of the feature itself. For more information, see Operating System Version.

Example

See Getting the System Version.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetVersion, OSVERSIONINFO, OSVERSIONINFOEX, VerifyVersionInfo

---

1.210 GetVolumeInformation

The **GetVolumeInformation** function retrieves information about a file system and volume whose root directory is specified.

```
GetVolumeInformation: procedure
(
        lpRootPathName:             string;
    var lpVolumeNameBuffer:         var;
        nVolumeNameSize:            dword;
    var lpVolumeSerialNumber:       dword;
    var lpMaximumComponentLength:   dword;
    var lpFileSystemFlags:          dword;
    var lpFileSystemNameBuffer:     var;
        nFileSystemNameSize:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetVolumeInformationA@32" );
```

Parameters

lpRootPathName

[in] Pointer to a string that contains the root directory of the volume to be described. If this parameter is NULL, the root of the current directory is used. A trailing backslash is required. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\, or the C drive as "C:\".

lpVolumeNameBuffer

[out] Pointer to a buffer that receives the name of the specified volume.

nVolumeNameSize

[in] Specifies the length, in **TCHARs**, of the volume name buffer. This parameter is ignored if the volume name buffer is not supplied.

lpVolumeSerialNumber

[out] Pointer to a variable that receives the volume serial number. This parameter can be NULL if the serial number is not required.

Windows 95/98: If the queried volume is a network drive, the serial number will not be returned.

lpMaximumComponentLength

[out] Pointer to a variable that receives the maximum length, in **TCHARs**, of a file name component supported by the specified file system. A file name component is that portion of a file name between backslashes.

The value stored in variable pointed to by ***lpMaximumComponentLength** is used to indicate that long names are supported by the specified file system. For example, for a FAT file system supporting long names, the function stores the value 255, rather than the previous 8.3 indicator. Long names can also be supported on systems that use the NTFS file system.

lpFileSystemFlags

[out] Pointer to a variable that receives flags associated with the specified file system. This parameter can be any combination of the following flags; however, FS_FILE_COMPRESSION and FS_VOL_IS_COMPRESSED are mutually exclusive.

Value	Meaning
FS_CASE_IS_PRESERVED	The file system preserves the case of file names when it places a name on disk.
FS_CASE_SENSITIVE	The file system supports case-sensitive file names.
FS_UNICODE_STORED_ON_DISK	The file system supports Unicode in file names as they appear on disk.
FS_PERSISTENT_ACLS	The file system preserves and enforces ACLs. For example, NTFS preserves and enforces ACLs, and FAT does not.
FS_FILE_COMPRESSION	The file system supports file-based compression.
FS_VOL_IS_COMPRESSED	The specified volume is a compressed volume; for example, a DoubleSpace volume.
FILE_NAMED_STREAMS	The file system supports named streams.
FILE_READ_ONLY_VOLUME	The specified volume is read-only.
FILE_SUPPORTS_ENCRYPTION	The file system supports the Encrypted File System (EFS).
FILE_SUPPORTS_OBJECT_IDS	The file system supports object identifiers.
FILE_SUPPORTS_REPARSE_POINTS	The file system supports reparse points.
FILE_SUPPORTS_SPARSE_FILES	The file system supports sparse files.
FILE_VOLUME_QUOTAS	The file system supports disk quotas.

lpFileSystemNameBuffer

[out] Pointer to a buffer that receives the name of the file system (such as FAT or NTFS).

nFileSystemNameSize

[in] Specifies the length, in **TCHARs**, of the file system name buffer. This parameter is ignored if the file system name buffer is not supplied.

Return Values

If all the requested information is retrieved, the return value is nonzero.

If not all the requested information is retrieved, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

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

The FS_VOL_IS_COMPRESSED flag is the only indicator of volume-based compression. The file system name is not altered to indicate compression. This flag comes back set on a DoubleSpace volume, for example. With volume-based compression, an entire volume is either compressed or not compressed.

The FS_FILE_COMPRESSION flag indicates whether a file system supports file-based compression. With file-based compression, individual files can be compressed or not compressed.

The FS_FILE_COMPRESSION and FS_VOL_IS_COMPRESSED flags are mutually exclusive; both bits cannot come back set.

The maximum component length value, stored in *lpMaximumComponentLength*, is the only indicator that a volume supports longer-than-normal FAT (or other file system) file names. The file system name is not altered to indicate support for long file names.

The **GetCompressedFileSize** function obtains the compressed size of a file. The **GetFileAttributes** function can determine whether an individual file is compressed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File Systems Overview, File System Functions, GetCompressedFileSize, GetFileAttributes, SetErrorMode, SetVolumeLabel

---

1.211 GetWindowsDirectory

The **GetWindowsDirectory** function retrieves the path of the Windows directory. The Windows directory contains such files as applications, initialization files, and help files.

```
GetWindowsDirectory: procedure
(
    var lpBuffer:   var;
        uSize:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GetWindowsDirectoryA@8" );
```

Parameters

lpBuffer

[out] Pointer to the buffer to receive the null-terminated string containing the path. This path does not end with a

backslash unless the Windows directory is the root directory. For example, if the Windows directory is named WINDOWS on drive C, the path of the Windows directory retrieved by this function is C:\WINDOWS. If the system was installed in the root directory of drive C, the path retrieved is C:\.

uSize

[in] Specifies the maximum size, in **TCHARs**, of the buffer specified by the *lpBuffer* parameter. This value should be set to MAX_PATH to allow sufficient room for the path.

Return Values

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to the buffer, not including the terminating null character.

If the length is greater than the size of the buffer, the return value is the size of the buffer required to hold the path.

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

Remarks

The Windows directory is the directory where an application should store initialization and help files. If the user is running a shared version of the system, the Windows directory is guaranteed to be private for each user.

If an application creates other files that it wants to store on a per-user basis, it should place them in the directory specified by the HOMEPATH environment variable. This directory will be different for each user, if so specified by an administrator, through the User Manager administrative tool. HOMEPATH always specifies either the user's home directory, which is guaranteed to be private for each user, or a default directory (for example, C:\USERS\DEFAULT) where the user will have all access.

Terminal Services: If the application is running in a Terminal Services environment, each user has a unique Windows directory. If an application that is not Terminal-Services-aware calls this function, it retrieves the path of the Windows directory on the client, not the Windows directory on the server.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetCurrentDirectory, GetSystemDirectory, GetSystemWindowsDirectory

---

1.212 GlobalAddAtom

The **GlobalAddAtom** function adds a character string to the global atom table and returns a unique value (an atom) identifying the string.

```
GlobalAddAtom: procedure
(
    lpString:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalAddAtomA@4" );
```

Parameters

lpString

[in] Pointer to the null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ only in case are considered identical. The case of the first string of this name added to the table is preserved and returned by the **GlobalGetAtomName** function.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the newly created atom.

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

Remarks

If the string already exists in the global atom table, the atom for the existing string is returned and the atom's reference count is incremented.

The string associated with the atom is not deleted from memory until its reference count is zero. For more information, see the **GlobalDeleteAtom** function.

Global atoms are not deleted automatically when the application terminates. For every call to the **GlobalAddAtom** function, there must be a corresponding call to the **GlobalDeleteAtom** function.

If the *lpString* parameter has the form "#1234", **GlobalAddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **GlobalAddAtom** returns a string atom.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GetAtomName, GlobalDeleteAtom, GlobalFindAtom, GlobalGetAtomName, MAKEINTATOM

1.213 GlobalAlloc

The **GlobalAlloc** function allocates the specified number of bytes from the heap. Win32 memory management does not provide a separate local heap and global heap.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalAlloc: procedure
(
    uFlags:     uns32;
    dwBytes:    SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalAlloc@8" );
```

Parameters

uFlags

[in] Specifies how to allocate memory. If zero is specified, the default is GMEM_FIXED. This parameter can be one or more of the following values, except for the incompatible combinations that are specifically noted.

Value	Meaning
GHND	Combines GMEM_MOVEABLE and GMEM_ZEROINIT.
GMEM_FIXED	Allocates fixed memory. The return value is a pointer.
GMEM_MOVEABLE	Allocates movable memory. In Win32, memory blocks are never moved in physical memory, but they can be moved within the default heap. The return value is a handle to the memory object. To translate the handle into a pointer, use the **GlobalLock** function. This value cannot be combined with GMEM_FIXED.
GMEM_ZEROINIT	Initializes memory contents to zero.
GPTR	Combines GMEM_FIXED and GMEM_ZEROINIT.

The following values are obsolete.

Value	Meaning
GMEM_DDESHARE	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **GlobalDiscard** function to discard a block. This value cannot be combined with GMEM_FIXED.
GMEM_LOWER	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_NOCOMPACT	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_NODISCARD	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_NOT_BANKED	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_NOTIFY	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_SHARE	Ignored. This value is provided only for compatibility with 16-bit Windows.

dwBytes

[in] Specifies the number of bytes to allocate. If this parameter is zero and the *uFlags* parameter specifies GMEM_MOVEABLE, the function returns a handle to a memory object that is marked as discarded.

Return Values

If the function succeeds, the return value is a handle to the newly allocated memory object.

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

Remarks

If the heap does not contain sufficient free space to satisfy the request, **GlobalAlloc** returns NULL. Because NULL is used to indicate an error, virtual address zero is never allocated. It is, therefore, easy to detect the use of a NULL pointer.

Memory allocated with this function is guaranteed to be aligned on an 8-byte boundary. All memory is created with execute access; no special function is required to execute dynamically generated code.

If this function succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual number of bytes allocated, use the **GlobalSize** function.

To free the memory, use the **GlobalFree** function.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalDiscard, GlobalFree, GlobalLock, GlobalSize

---

1.214 GlobalDeleteAtom

The **GlobalDeleteAtom** function decrements the reference count of a global string atom. If the atom's reference count reaches zero, **GlobalDeleteAtom** removes the string associated with the atom from the global atom table.

```
GlobalDeleteAtom: procedure
(
    nAtom:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalDeleteAtom@4" );
```

Parameters

nAtom

[in] Identifies the atom and character string to be deleted.

Return Values

If the function succeeds, the return value is zero.

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

Remarks

A string atom's reference count specifies the number of times the string has been added to the atom table. The **GlobalAddAtom** function increments the reference count of a string that already exists in the global atom table each time it is called.

Each call to **GlobalAddAtom** should have a corresponding call to **GlobalDeleteAtom**. Do not call **GlobalDeleteAtom** more times than you call **GlobalAddAtom**, or you may delete the atom while other clients are using it. Applications using DDE should follow the rules on global atom management to prevent leaks and premature deletion.

GlobalDeleteAtom has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GlobalAddAtom, GlobalFindAtom, MAKEINTATOM

1.215 GlobalFindAtom

The **GlobalFindAtom** function searches the global atom table for the specified character string and retrieves the global atom associated with that string.

```
GlobalFindAtom: procedure
(
    lpString:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalFindAtomA@4" );
```

Parameters

lpString

[in] Pointer to the null-terminated character string for which to search.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the global atom associated with the given string.

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

Remarks

Even though the system preserves the case of a string in an atom table as it was originally entered, the search performed by **GlobalFindAtom** is not case sensitive.

If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GetAtomName, GlobalAddAtom, GlobalDeleteAtom, GlobalGetAtomName

---

1.216 GlobalFlags

The **GlobalFlags** function returns information about the specified global memory object.

Note This function is provided only for compatibility with 16-bit versions of Windows.

```
GlobalFlags: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalFlags@4" );
```

Parameters

hMem

[in] Handle to the global memory object. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

Return Values

If the function succeeds, the return value specifies the allocation values and the lock count for the memory object.

If the function fails, the return value is GMEM_INVALID_HANDLE, indicating that the global handle is not valid. To get extended error information, call **GetLastError**.

Remarks

The low-order byte of the low-order word of the return value contains the lock count of the object. To retrieve the lock count from the return value, use the GMEM_LOCKCOUNT mask with the bitwise AND (&) operator. The lock count of memory objects allocated with GMEM_FIXED is always zero.

The high-order byte of the low-order word of the return value indicates the allocation values of the memory object. It can be zero or any combination of the following values.

Value	Meaning
GMEM_DDESHARE	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **GlobalDiscard** function to discard a block.
GMEM_DISCARDED	The object's memory block has been discarded.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalDiscard, GlobalReAlloc

---

1.217 GlobalFree

The **GlobalFree** function frees the specified global memory object and invalidates its handle.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalFree: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalFree@4" );
```

Parameters

hMem

[in] Handle to the global memory object. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

Return Values

If the function succeeds, the return value is NULL.

If the function fails, the return value is equal to a handle to the global memory object. To get extended error information, call **GetLastError**.

Remarks

If the process examines or modifies the memory after it has been freed, heap corruption may occur or an access violation exception (EXCEPTION_ACCESS_VIOLATION) may be generated.

If the *hgblMem* parameter is NULL, **GlobalFree** fails and the system generates an access violation exception.

The **GlobalFree** function will free a locked memory object. A locked memory object has a lock count greater than zero. The **GlobalLock** function locks a global memory object and increments the lock count by one. The **GlobalUnlock** function unlocks it and decrements the lock count by one. To get the lock count of a global memory object, use the **GlobalFlags** function.

If an application is running under a debug version of the system, **GlobalFree** will issue a message that tells you that a locked object is being freed. If you are debugging the application, **GlobalFree** will enter a breakpoint just before freeing a locked object. This allows you to verify the intended behavior, then continue execution.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalFlags, GlobalLock, GlobalReAlloc, GlobalUnlock

---

1.218 GlobalGetAtomName

The **GlobalGetAtomName** function retrieves a copy of the character string associated with the specified global atom.

```
GlobalGetAtomName: procedure
(
        nAtom:      dword;
    var lpBuffer:   var;
        nSize:      dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalGetAtomNameA@12" );
```

Parameters

nAtom

[in] Identifies the global atom associated with the character string to be retrieved.

lpBuffer

[out] Pointer to the buffer for the character string.

nSize

[in] Specifies the size, in **TCHARs**, of the buffer. For ANSI versions of the function this is the number of bytes, while for wide-character (Unicode) versions this is the number of characters.

Return Values

If the function succeeds, the return value is the length of the string copied to the buffer, in **TCHARs**, not including the terminating null character.

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

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GlobalAddAtom, GlobalDeleteAtom, GlobalFindAtom, MAKEINTATOM

1.219 GlobalHandle

The **GlobalHandle** function retrieves the handle associated with the specified pointer to a global memory block.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalHandle: procedure
(
    var pMem:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalHandle@4" );
```

Parameters

pMem

[in] Pointer to the first byte of the global memory block. This pointer is returned by the **GlobalLock** function.

Return Values

If the function succeeds, the return value is a handle to the specified global memory object.

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

Remarks

When the **GlobalAlloc** function allocates a memory object with GMEM_MOVEABLE, it returns a handle to the object. The **GlobalLock** function converts this handle into a pointer to the memory block, and **GlobalHandle** converts the pointer back into a handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalLock

1.220 GlobalLock

The **GlobalLock** function locks a global memory object and returns a pointer to the first byte of the object's memory block.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalLock: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalLock@4" );
```

Parameters

hMem

[in] Handle to the global memory object. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

Return Values

If the function succeeds, the return value is a pointer to the first byte of the memory block.

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

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, **GlobalLock** increments the count by one, and the **GlobalUnlock** function decrements the count by one. For each call that a process makes to **GlobalLock** for an object, it must eventually call **GlobalUnlock**. Locked memory will not be moved or discarded, unless the memory object is reallocated by using the **GlobalReAlloc** function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with GMEM_FIXED always have a lock count of zero. For these objects, the value of the returned pointer is equal to the value of the specified handle.

If the specified memory block has been discarded or if the memory block has a zero-byte size, this function returns NULL.

Discarded objects always have a lock count of zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalReAlloc, GlobalUnlock

---

1.221 GlobalMemoryStatus

The **GlobalMemoryStatus** function obtains information about the system's current usage of both physical and virtual memory.

To obtain information about the extended portion of the virtual address space, or if your application may run on computers with more than 4 GB of main memory, use the **GlobalMemoryStatusEx** function.

```
GlobalMemoryStatus: procedure
(
    var lpBuffer:   MEMORYSTATUS
);
    stdcall;
    external( "__imp__GlobalMemoryStatus@4" );
```

Parameters

lpBuffer

[out] Pointer to a **MEMORYSTATUS** structure. The **GlobalMemoryStatus** function stores information about current

memory availability into this structure.

Return Values

This function does not return a value.

Remarks

You can use the **GlobalMemoryStatus** function to determine how much memory your application can allocate without severely impacting other applications.

The information returned by the **GlobalMemoryStatus** function is volatile. There is no guarantee that two sequential calls to this function will return the same information.

On computers with more than 4 GB of memory, the **GlobalMemoryStatus** function can return incorrect information. Windows 2000 reports a value of -1 to indicate an overflow. Earlier versions of Windows NT report a value that is the real amount of memory, modulo 4 GB. For this reason, on Windows 2000, use the **GlobalMemoryStatusEx** function instead.

On Intel x86 computers with more than 2 GB and less than 4 GB of memory, the **GlobalMemoryStatus** function will always return 2 GB in the **dwTotalPhys** member of the **MEMORYSTATUS** structure. Similarly, if the total available memory is between 2 and 4 GB, the **dwAvailPhys** member of the **MEMORYSTATUS** structure will be rounded down to 2 GB. If the executable is linked using the /LARGEADDRESSWARE linker option, then the **GlobalMemoryStatus** function will return the correct amount of physical memory in both members.

Example

The program following shows a simple use of the **GlobalMemoryStatus** function.

```
//  Sample output:
//  c:\>global
//  The MemoryStatus structure is 32 bytes long.
//  It should be 32.
//  78 percent of memory is in use.
//  There are   65076 total Kbytes of physical memory.
//  There are   13756 free Kbytes of physical memory.
//  There are  150960 total Kbytes of paging file.
//  There are   87816 free Kbytes of paging file.
//  There are  1fff80 total Kbytes of virtual memory.
//  There are  1fe770 free Kbytes of virtual memory.

#include <windows.h>

// Use to change the divisor from Kb to Mb.

#define DIV 1024
// #define DIV 1

char *divisor = "K";
// char *divisor = "";

// Handle the width of the field in which to print numbers this way to
// make changes easier. The asterisk in the print format specifier
// "%*ld" takes an int from the argument list, and uses it to pad and
// right-justify the number being formatted.
#define WIDTH 7

void main(int argc, char *argv[])
{
  MEMORYSTATUS stat;
```

```
  GlobalMemoryStatus (&stat);

  printf ("The MemoryStatus structure is %ld bytes long.\n",
          stat.dwLength);
  printf ("It should be %d.\n", sizeof (stat));
  printf ("%ld percent of memory is in use.\n",
          stat.dwMemoryLoad);
  printf ("There are %*ld total %sbytes of physical memory.\n",
          WIDTH, stat.dwTotalPhys/DIV, divisor);
  printf ("There are %*ld free %sbytes of physical memory.\n",
          WIDTH, stat.dwAvailPhys/DIV, divisor);
  printf ("There are %*ld total %sbytes of paging file.\n",
          WIDTH, stat.dwTotalPageFile/DIV, divisor);
  printf ("There are %*ld free %sbytes of paging file.\n",
          WIDTH, stat.dwAvailPageFile/DIV, divisor);
  printf ("There are %*lx total %sbytes of virtual memory.\n",
          WIDTH, stat.dwTotalVirtual/DIV, divisor);
  printf ("There are %*lx free %sbytes of virtual memory.\n",
          WIDTH, stat.dwAvailVirtual/DIV, divisor);
}
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalMemoryStatusEx, MEMORYSTATUS

1.222 GlobalMemoryStatusEx

The **GlobalMemoryStatusEx** function obtains information about the system's current usage of both physical and virtual memory.

```
GlobalMemoryStatusVlm: procedure
(
    var lpBuffer:   MEMORYSTATUSEX
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalMemoryStatusVlm@4" );
```

Parameters

lpBuffer

[in/out] Pointer to a **MEMORYSTATUSEX** structure. **GlobalMemoryStatusEx** stores information about current memory availability in this structure.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

You can use the **GlobalMemoryStatusEx** function to determine how much memory your application can allocate without severely impacting other applications.

The information returned by the **GlobalMemoryStatusEx** function is volatile. There is no guarantee that two sequential calls to this function will return the same information.

Example

The program following shows a simple use of the **GlobalMemoryStatusEx** function.

```
//  Sample output:

//  c:\>globalex
//  78 percent of memory is in use.
//  There are   65076 total Kbytes of physical memory.
//  There are   14248 free Kbytes of physical memory.
//  There are  150960 total Kbytes of paging file.
//  There are   88360 free Kbytes of paging file.
//  There are  1fff80 total Kbytes of virtual memory.
//  There are  1fe770 free Kbytes of virtual memory.
//  There are       0 free Kbytes of extended memory.

#define _WIN32_WINNT 0x0500

#include <windows.h>

// Use to change the divisor from Kb to Mb.

#define DIV 1024
// #define DIV 1

char *divisor = "K";
// char *divisor = "";

// Handle the width of the field in which to print numbers this way to
// make changes easier. The asterisk in the print format specifier
// "%*I64d" takes an int from the argument list, and uses it to pad
// and right-justify the number being formatted.
#define WIDTH 7

void main(int argc, char *argv[])
{
  MEMORYSTATUSEX statex;

  statex.dwLength = sizeof (statex);

  GlobalMemoryStatusEx (&statex);

  printf ("%ld percent of memory is in use.\n",
          statex.dwMemoryLoad);
  printf ("There are %*I64d total %sbytes of physical memory.\n",
          WIDTH, statex.ullTotalPhys/DIV, divisor);
  printf ("There are %*I64d free %sbytes of physical memory.\n",
          WIDTH, statex.ullAvailPhys/DIV, divisor);
  printf ("There are %*I64d total %sbytes of paging file.\n",
          WIDTH, statex.ullTotalPageFile/DIV, divisor);
  printf ("There are %*I64d free %sbytes of paging file.\n",
```

```
        WIDTH, statex.ullAvailPageFile/DIV, divisor);
  printf ("There are %*I64x total %sbytes of virtual memory.\n",
        WIDTH, statex.ullTotalVirtual/DIV, divisor);
  printf ("There are %*I64x free %sbytes of virtual memory.\n",
        WIDTH, statex.ullAvailVirtual/DIV, divisor);

  // Show the amount of extended memory available.

  printf ("There are %*I64x free %sbytes of extended memory.\n",
        WIDTH, statex.ullAvailExtendedVirtual/DIV, divisor);

}
```

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, MEMORYSTATUSEX

1.223 GlobalReAlloc

The **GlobalReAlloc** function changes the size or attributes of a specified global memory object. The size can increase or decrease.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalReAlloc: procedure
(
    hMem:       dword;
    dwBytes:    SIZE_T;
    uFlags:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalReAlloc@12" );
```

Parameters

hMem

[in] Handle to the global memory object to be reallocated. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

dwBytes

[in] Specifies the new size, in bytes, of the memory block. If *uFlags* specifies GMEM_MODIFY, this parameter is ignored.

uFlags

[in] Specifies how to reallocate the global memory object. If GMEM_MODIFY is specified, this parameter modifies the attributes of the memory object, and the *dwBytes* parameter is ignored. Otherwise, this parameter controls the reallocation of the memory object.

You can combine GMEM_MODIFY with one or both of the following values.

Value	Meaning
GMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **GlobalDiscard** function to discard a block. In 16-bit Windows, allocates discardable memory, if GMEM_MODIFY is also specified. This value is ignored if the object was not previously allocated as movable or if GMEM_MOVEABLE is not specified.
GMEM_MOVEABLE	If GMEM_MODIFY is also specified, GMEM_MOVEABLE changes a fixed memory object to a movable memory object. If GMEM_MODIFY is not specified, then GMEM_MOVEABLE allows a locked GMEM_MOVEABLE memory block or a GMEM_FIXED memory block to be moved to a new fixed location. If neither GMEM_MODIFY nor GMEM_MOVEABLE is set, then fixed memory blocks and locked movable memory blocks will only be reallocated in place.

If this parameter does not specify GMEM_MODIFY, it can also be any combination of the following values.

Value	Meaning
GMEM_NOCOMPACT	Ignored. This value is provided only for compatibility with 16-bit Windows.
GMEM_ZEROINIT	Causes the additional memory contents to be initialized to zero if the memory object is growing in size.

Return Values

If the function succeeds, the return value is a handle to the reallocated memory object.

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

Remarks

If **GlobalReAlloc** reallocates a movable object, the return value is a handle to the memory object. To convert the handle to a pointer, use the **GlobalLock** function.

If **GlobalReAlloc** reallocates a fixed object, the value of the handle returned is the address of the first byte of the memory block. To access the memory, a process can simply cast the return value to a pointer.

If **GlobalReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalDiscard, GlobalLock

1.224 GlobalSize

The **GlobalSize** function retrieves the current size, in bytes, of the specified global memory object.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalSize: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__GlobalSize@4" );
```

Parameters

hMem

[in] Handle to the global memory object. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

Return Values

If the function succeeds, the return value is the size, in bytes, of the specified global memory object.

If the specified handle is not valid or if the object has been discarded, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The size of a memory block may be larger than the size requested when the memory was allocated.

To verify that the specified object's memory block has not been discarded, use the **GlobalFlags** function before calling **GlobalSize**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalFlags, GlobalReAlloc

1.225 GlobalUnlock

The **GlobalUnlock** function decrements the lock count associated with a memory object that was allocated with GMEM_MOVEABLE. This function has no effect on memory objects allocated with GMEM_FIXED.

Note The global functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions. However, the global functions are still used with DDE and the clipboard functions.

```
GlobalUnlock: procedure
(
    hMem:dword
);
```

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

Parameters

hMem

[in] Handle to the global memory object. This handle is returned by either the **GlobalAlloc** or **GlobalReAlloc** function.

Return Values

If the memory object is still locked after decrementing the lock count, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. If **GetLastError** returns NO_ERROR, the memory object is unlocked.

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, the **GlobalLock** function increments the count by one, and **GlobalUnlock** decrements the count by one. For each call that a process makes to **GlobalLock** for an object, it must eventually call **GlobalUnlock**. Locked memory will not be moved or discarded, unless the memory object is reallocated by using the **GlobalReAlloc** function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with GMEM_FIXED always have a lock count of zero. If the specified memory block is fixed memory, this function returns TRUE.

If the memory object is already unlocked, **GlobalUnlock** returns FALSE and **GetLastError** reports ERROR_NOT_LOCKED.

A process should not rely on the return value to determine the number of times it must subsequently call **GlobalUnlock** for a memory object.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalAlloc, GlobalLock, GlobalReAlloc

---

1.226 Heap32First

Retrieves information about the first block of a heap that has been allocated by a process.

```
Heap32First: procedure
(
    var lphe:           HEAPENTRY32;
        th32ProcessID:  dword;
    var th32HeapID:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__Heap32First@12" );
```

Parameters

lphe

[in/out] Pointer to a **HEAPENTRY32** structure.

th32ProcessID

[in] Identifier of the process context that owns the heap.

th32HeapID

[in] Identifier of the heap to enumerate.

Return Values

Returns TRUE if information for the first heap block has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if the heap is invalid or empty.

Remarks

The calling application must set the **dwSize** member of **HEAPENTRY32** to the size, in bytes, of the structure. **Heap32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To access subsequent blocks of the same heap, use the **Heap32Next** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions, , Heap32Next, HEAPENTRY32

1.227 Heap32ListFirst

Retrieves information about the first heap that has been allocated by a specified process.

```
Heap32ListFirst: procedure
(
        hSnapshot:  dword;
    var lphl:       HEAPLIST32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Heap32ListFirst@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lphl

[in/out] Pointer to a **HEAPLIST32** structure.

Return Values

Returns TRUE if the first entry of the heap list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function when no heap list exists or the snapshot does not contain heap list information.

Remarks

The calling application must set the **dwSize** member of **HEAPLIST32** to the size, in bytes, of the structure. **Heap32ListFirst** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other heaps in the heap list, use the **Heap32ListNext** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions, , CreateToolhelp32Snapshot, HEAPLIST32, Heap32ListNext

---

1.228 Heap32ListNext

Retrieves information about the next heap that has been allocated by a process.

```
Heap32ListNext: procedure
(
        hSnapshot:  dword;
    var lphl:       HEAPLIST32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Heap32ListNext@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lphl

[out] Pointer to a **HEAPLIST32** structure.

Return Values

Returns TRUE if the next entry of the heap list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function when no more entries in the heap list exist.

Remarks

To retrieve information about the first heap in a heap list, use the **Heap32ListFirst** function.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.229 Heap32Next

Retrieves information about the next block of a heap that has been allocated by a process.

```
Heap32Next: procedure
(
    var lphe:   HEAPENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Heap32Next@4" );
```

Parameters

lphe

[out] Pointer to a **HEAPENTRY32** structure.

Return Values

Returns TRUE if information about the next block in the heap has been copied to the buffer or FALSE otherwise. The **GetLastError** function returns ERROR_NO_MORE_FILES when no more objects in the heap exist and ERROR_INVALID_DATA if the heap appears to be corrupt or is modified during the walk in such a way that **Heap32Next** cannot continue.

Remarks

To retrieve information for the first block of a heap, use the **Heap32First** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.230 HeapAlloc

The **HeapAlloc** function allocates a block of memory from a heap. The allocated memory is not movable.

```
HeapAlloc: procedure
(
    hHeap:      dword;
    dwFlags:    dword;
    dwBytes:    SIZE_T
);
```

```
stdcall;
returns( "eax" );
external( "__imp__HeapAlloc@12" );
```

Parameters

hHeap

[in] Specifies the heap from which the memory will be allocated. This parameter is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of heap allocation. Specifying any of these values will override the corresponding value specified when the heap was created with **HeapCreate**. This parameter can be one or more of the following values.

Value	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the system will raise an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while the **HeapAlloc** function is accessing the heap. This value should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.
HEAP_ZERO_MEMORY	Specifies that the allocated memory will be initialized to zero. Otherwise, the memory is not initialized to zero.

dwBytes

[in] Specifies the number of bytes to be allocated.

If the heap specified by the *hHeap* parameter is a "non-growable" heap, dwBytes must be less than 0x7FFF8. You create a non-growable heap by calling the HeapCreate function with a nonzero value.

Return Values

If the function succeeds, the return value is a pointer to the allocated memory block.

If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.

If the function fails and you have specified HEAP_GENERATE_EXCEPTIONS, the function may generate the following exceptions:

Value	Meaning
STATUS_NO_MEMORY	The allocation attempt failed because of a lack of available memory or heap corruption.
STATUS_ACCESS_VIOLATION	The allocation attempt failed because of heap corruption or improper function parameters.

Note Heap corruption can lead to either exception. It depends upon the nature of the heap corruption.

If the function fails, it does not call **SetLastError**. An application cannot call **GetLastError** for extended error information.

Remarks

If **HeapAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the allocated block, use the **HeapSize** function.

To free a block of memory allocated by **HeapAlloc**, use the **HeapFree** function.

Memory allocated by **HeapAlloc** is not movable. Since the memory is not movable, it is possible for the heap to become fragmented.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapCreate, HeapDestroy, HeapFree, HeapReAlloc, HeapSize, SetLastError

---

1.231 HeapCompact

The **HeapCompact** function attempts to compact a specified heap. It compacts the heap by coalescing adjacent free blocks of memory and decommitting large free blocks of memory.

```
HeapCompact: procedure
(
    hHeap:      dword;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapCompact@8" );
```

Parameters

hHeap

[in] Handle to the heap that the function will attempt to compact.

dwFlags

[in] Specifies heap access during function operation. This parameter can be the following value.

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while the **HeapCompact** function accesses the heap.

Return Values

If the function succeeds, the return value is the size, in bytes, of the largest committed free block in the heap. This is an unsigned integer value.

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

In the unlikely case that there is absolutely no space available in the heap, the function return value is zero, and **GetLastError** returns the value NO_ERROR.

Remarks

There is no guarantee that an application can successfully allocate a memory block of the size returned by **HeapCompact**. Other threads or the commit threshold might prevent such an allocation.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapCreate, HeapValidate

---

1.232 HeapCreate

The **HeapCreate** function creates a heap object that can be used by the calling process. The function reserves space in the virtual address space of the process and allocates physical storage for a specified initial portion of this block.

```
HeapCreate: procedure
(
    flOptions:      dword;
    dwInitialSize:  SIZE_T;
```

```
    dwMaximumSize:  SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapCreate@12" );
```

Parameters

flOptions

[in] Specifies optional attributes for the new heap. These options affect subsequent access to the new heap through calls to the heap functions (**HeapAlloc, HeapFree, HeapReAlloc**, and **HeapSize**). You can specify one or more of the following values.

Value	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the system will raise an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used when the heap functions allocate and free memory from this heap. The default, when HEAP_NO_SERIALIZE is not specified, is to serialize access to the heap. Serialization of heap access allows two or more threads to simultaneously allocate and free memory from the same heap.

dwInitialSize

[in] Specifies the initial size, in bytes, of the heap. This value determines the initial amount of physical storage that is allocated for the heap. The value is rounded up to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

dwMaximumSize

[in] If *dwMaximumSize* is a nonzero value, it specifies the maximum size, in bytes, of the heap. The **HeapCreate** function rounds *dwMaximumSize* up to the next page boundary, and then reserves a block of that size in the process's virtual address space for the heap. If allocation requests made by the **HeapAlloc** or **HeapReAlloc** functions exceed the initial amount of physical storage specified by *dwInitialSize*, the system allocates additional pages of physical storage for the heap, up to the heap's maximum size.

In addition, if *dwMaximumSize* is nonzero, the heap cannot grow, and an absolute limitation arises: the maximum size of a memory block in the heap is a bit less than 0x7FFF8 bytes. Requests to allocate larger blocks will fail, even if the maximum size of the heap is large enough to contain the block.

If *dwMaximumSize* is zero, it specifies that the heap is growable. The heap's size is limited only by available memory. Requests to allocate blocks larger than 0x7FFF8 bytes do not automatically fail; the system calls **VirtualAlloc** to obtain the memory needed for such large blocks. Applications that need to allocate large memory blocks should set *dwMaximumSize* to zero.

Return Values

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

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

Remarks

The **HeapCreate** function creates a private heap object from which the calling process can allocate memory blocks by using the **HeapAlloc** function. The initial size determines the number of committed pages that are initially allocated for the heap. The maximum size determines the total number of reserved pages. These pages create a block in the process's virtual address space into which the heap can grow. If requests by **HeapAlloc** exceed the current size of

committed pages, additional pages are automatically committed from this reserved space, assuming that the physical storage is available.

The memory of a private heap object is accessible only to the process that created it. If a dynamic-link library (DLL) creates a private heap, the heap is created in the address space of the process that called the DLL, and it is accessible only to that process.

The system uses memory from the private heap to store heap support structures, so not all of the specified heap size is available to the process. For example, if the **HeapAlloc** function requests 64 kilobytes (K) from a heap with a maximum size of 64K, the request may fail because of system overhead.

If HEAP_NO_SERIALIZE is not specified (the simple default), the heap will serialize access within the calling process. Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap.

Setting HEAP_NO_SERIALIZE eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. Therefore, HEAP_NO_SERIALIZE can safely be used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, GetProcessHeaps, GetSystemInfo, HeapAlloc, HeapDestroy, HeapFree, HeapReAlloc, HeapSize, HeapValidate, VirtualAlloc

---

1.233 HeapDestroy

The **HeapDestroy** function destroys the specified heap object. **HeapDestroy** decommits and releases all the pages of a private heap object, and it invalidates the handle to the heap.

```
HeapDestroy: procedure
(
    hHeap:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapDestroy@4" );
```

Parameters

hHeap

[in] Specifies the heap to be destroyed. This parameter should be a heap handle returned by the **HeapCreate** function. Do not use the handle to the process heap returned by the **GetProcessHeap** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Processes can call **HeapDestroy** without first calling the **HeapFree** function to free memory allocated from the heap.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapAlloc, HeapCreate, HeapFree, HeapReAlloc, HeapSize

---

1.234 HeapFree

The **HeapFree** function frees a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

```
HeapFree: procedure
(
        hHeap:      dword;
        dwFlags:    dword;
    var lpMem:      var
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapFree@12" );
```

Parameters

hHeap

[in] Specifies the heap whose memory block the function frees. This parameter is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of freeing a memory block. Specifying the following value overrides the corresponding value specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function.

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while **HeapFree** is accessing the heap. Do not specify this value when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.

lpMem

[in] Pointer to the memory block to free. This pointer is returned by the **HeapAlloc** or **HeapReAlloc** function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. An application can call **GetLastError** for extended error information.

Remarks

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

You should not refer in any way to memory that has been freed by **HeapFree**. Once that memory is freed, any information that may have been in it is gone forever. If you require information, do not free memory containing the information. Function calls that return information about memory (such as **HeapSize**) may not be used with freed memory, as they may return bogus data.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapAlloc, HeapCreate, HeapDestroy, HeapReAlloc, HeapSize, SetLastError

---

1.235 HeapLock

The **HeapLock** function attempts to acquire the critical section object, or lock, that is associated with a specified heap.

If the function succeeds, the calling thread owns the heap lock. Only the calling thread will be able to allocate or release memory from the heap. The execution of any other thread of the calling process will be blocked if that thread attempts to allocate or release memory from the heap. Such threads will remain blocked until the thread that owns the heap lock calls the **HeapUnlock** function.

```
HeapLock: procedure
(
    hHeap:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapLock@4" );
```

Parameters

hHeap

[in] Handle to the heap to lock for exclusive access by the calling thread.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **HeapLock** function is primarily useful for preventing the allocation and release of heap memory by other threads while the calling thread uses the **HeapWalk** function.

Each call to **HeapLock** must be matched by a corresponding call to the **HeapUnlock** function. Failure to call **HeapUnlock** will block the execution of any other threads of the calling process that attempt to access the heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapUnlock, HeapWalk

---

1.236 HeapReAlloc

The **HeapReAlloc** function reallocates a block of memory from a heap. This function enables you to resize a memory block and change other memory block properties. The allocated memory is not movable.

```
HeapReAlloc: procedure
(
        hHeap:      dword;
        dwFlags:    dword;
    var lpMem:      var;
        dwBytes:    SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapReAlloc@16" );
```

Parameters

hHeap

[in] Heap from which the memory will be reallocated. This is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of heap reallocation. Specifying a value overrides the corresponding value specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function. This

parameter can be one or more of the following values.

Value	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the operating-system raises an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion is not used while **HeapReAlloc** is accessing the heap. This value should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.
HEAP_REALLOC_IN_PLACE_ONLY	Specifies that there can be no movement when reallocating a memory block to a larger size. If this value is not specified and the reallocation request is for a larger size, the function may move the block to a new location. If this value is specified and the block cannot be enlarged without moving, the function fails, leaving the original memory block unchanged.
HEAP_ZERO_MEMORY	If the reallocation request is for a larger size, this value specifies that the additional region of memory beyond the original size be initialized to zero. The contents of the memory block up to its original size are unaffected.

lpMem

[in] Pointer to the block of memory that the function reallocates. This pointer is returned by an earlier call to the **HeapAlloc** or **HeapReAlloc** function.

dwBytes

[in] New size of the memory block, in bytes. A memory block's size can be increased or decreased by using this function.

If the heap specified by the *hHeap* parameter is a "non-growable" heap, *dwBytes* must be less than 0x7FFF8. You create a non-growable heap by calling the **HeapCreate** function with a nonzero value.

Return Values

If the function succeeds, the return value is a pointer to the reallocated memory block.

If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.

If the function fails and you have specified HEAP_GENERATE_EXCEPTIONS, the function may generate the following exceptions:

Value	Meaning
STATUS_NO_MEMORY	The reallocation attempt failed for lack of available memory.
STATUS_ACCESS_VIOLATION	The reallocation attempt failed because of heap corruption or improper function parameters.

If the function fails, it calls **SetLastError**. An application can call **GetLastError** for extended error information.

Remarks

If **HeapReAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the reallo-

cated block, use the **HeapSize** function.

If **HeapReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

To free a block of memory allocated by **HeapReAlloc**, use the **HeapFree** function.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapAlloc, HeapCreate, HeapDestroy, HeapFree, HeapSize, SetLastError

---

1.237 HeapSize

The **HeapSize** function returns the size, in bytes, of a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

```
HeapSize: procedure
(
        hHeap:      dword;
        dwFlags:    dword;
    var lpMem:      var
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapSize@12" );
```

Parameters

hHeap

[in] Specifies the heap in which the memory block resides. This handle is returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of accessing the memory block. Specifying the following value overrides the corresponding value specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function.

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while **HeapSize** is accessing the heap. This value should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.

lpMem

[in] Pointer to the memory block whose size the function will obtain. This is a pointer returned by the **HeapAlloc** or **HeapReAlloc** function.

Return Values

If the function succeeds, the return value is the size, in bytes, of the allocated memory block.

If the function fails, the return value is -1. The function does not call **SetLastError**. An application cannot call **GetLastError** for extended error information.

Remarks

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetProcessHeap, HeapAlloc, HeapCreate, HeapDestroy, HeapFree, HeapReAlloc, SetLastError

---

1.238 HeapUnlock

The **HeapUnlock** function releases ownership of the critical section object, or lock, that is associated with a specified heap. The **HeapUnlock** function reverses the action of the **HeapLock** function.

```
HeapUnlock: procedure
```

```
(
    hHeap:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapUnlock@4" );
```

Parameters

hHeap

[in] Handle to the heap to unlock.

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 **HeapLock** function is primarily useful for preventing the allocation and release of heap memory by other threads while the calling thread uses the **HeapWalk** function. The **HeapUnlock** function is the inverse of **HeapLock**.

Each call to **HeapLock** must be matched by a corresponding call to the **HeapUnlock** function. Failure to call **HeapUnlock** will block the execution of any other threads of the calling process that attempt to access the heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapLock, HeapWalk

---

1.239 HeapValidate

The **HeapValidate** function attempts to validate a specified heap. The function scans all the memory blocks in the heap, and verifies that the heap control structures maintained by the heap manager are in a consistent state. You can also use the **HeapValidate** function to validate a single memory block within a specified heap, without checking the validity of the entire heap.

```
HeapValidate: procedure
(
        hHeap:      dword;
        dwFlags:    dword;
    var lpMem:      var
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapValidate@12" );
```

Parameters

hHeap

[in] Handle to the heap of interest. The **HeapValidate** function attempts to validate this heap, or a single memory block within this heap.

dwFlags

[in] Specifies heap access during function operation. This parameter can be the following value.

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion is not used while the **HeapValidate** function accesses the heap.

lpMem

[in] Pointer to a memory block within the specified heap. This parameter may be NULL.

If this parameter is NULL, the function attempts to validate the entire heap specified by *hHeap*.

If this parameter is not NULL, the function attempts to validate the memory block pointed to by *lpMem*. It does not attempt to validate the rest of the heap.

Return Values

If the specified heap or memory block is valid, the return value is nonzero.

If the specified heap or memory block is invalid, the return value is zero. On a system set up for debugging, the **HeapValidate** function then displays debugging messages that describe the part of the heap or memory block that is invalid, and stops at a hard-coded breakpoint so that you can examine the system to determine the source of the invalidity. The **HeapValidate** function does not set the thread's last error value. There is no extended error information for this function; do not call **GetLastError**.

Remarks

There are heap control structures for each memory block in a heap, and for the heap as a whole. When you use the **HeapValidate** function to validate a complete heap, it checks all of these control structures for consistency.

When you use **HeapValidate** to validate a single memory block within a heap, it checks only the control structures pertaining to that element. **HeapValidate** can only validate allocated memory blocks. Calling **HeapValidate** on a freed memory block will return FALSE because there are no control structures to validate.

If you want to validate the heap elements enumerated by the **HeapWalk** function, you should only call **HeapValidate** on the elements that have PROCESS_HEAP_ENTRY_BUSY in the **wFlags** member of the **PROCESS_HEAP_ENTRY** structure. **HeapValidate** returns FALSE for all heap elements that do not have this bit set.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE value eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE value can, therefore, be safely used only in the following situations:

The process has only one thread.

The process has multiple threads, but only one thread calls the heap functions for a specific heap.

The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Validating a heap may degrade performance, especially on symmetric multiprocessing (SMP) computers. The side effects may last until the process ends.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or

reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapCreate, HeapWalk, PROCESS_HEAP_ENTRY

1.240 HeapWalk

The **HeapWalk** function enumerates the memory blocks in a specified heap created or manipulated by Win32 heap memory allocators such as **HeapAlloc, HeapReAlloc**, and **HeapFree**.

```
HeapWalk: procedure
(
        hHeap:      dword;
    var lpEntry:    PROCESS_HEAP_ENTRY
);
    stdcall;
    returns( "eax" );
    external( "__imp__HeapWalk@8" );
```

Parameters

hHeap

[in] Handle to the heap whose memory blocks you wish to enumerate.

lpEntry

[in/out] Pointer to a **PROCESS_HEAP_ENTRY** structure that maintains state information for a particular heap enumeration.

If the **HeapWalk** function succeeds, returning the value TRUE, this structure's members contain information about the next memory block in the heap.

To initiate a heap enumeration, set the **lpData** field of the **PROCESS_HEAP_ENTRY** structure to NULL. To continue a particular heap enumeration, call the **HeapWalk** function repeatedly, with no changes to *hHeap*, *lpEntry*, or any of the members of the **PROCESS_HEAP_ENTRY** structure.

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 heap enumeration terminates successfully by reaching the end of the heap, the function returns FALSE, and **GetLastError** returns the error code ERROR_NO_MORE_ITEMS.

Remarks

To initiate a heap enumeration, call **HeapWalk** with the **lpData** field of the **PROCESS_HEAP_ENTRY** structure pointed to by *lpEntry* set to NULL.

To continue a heap enumeration, call **HeapWalk** with the same *hHeap* and *lpEntry* values, and with the **PROCESS_HEAP_ENTRY** structure unchanged from the preceding call to **HeapWalk**. Repeat this process until

you have no need for further enumeration, or until the function returns FALSE and **GetLastError** returns ERROR_NO_MORE_ITEMS, indicating that all of the heap's memory blocks have been enumerated.

No special call of **HeapWalk** is needed to terminate the heap enumeration, since no enumeration state data is maintained outside the contents of the **PROCESS_HEAP_ENTRY** structure.

HeapWalk can fail in a multithreaded application if the heap is not locked during the heap enumeration. Use the **HeapLock** and **HeapUnlock** functions to control heap locking during heap enumeration.

Walking a heap may degrade performance, especially on symmetric multiprocessing (SMP) computers. The side effects may last until the process ends.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see Reading and Writing and Structured Exception Handling.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapAlloc, HeapReAlloc, HeapFree, HeapLock, HeapUnlock, HeapValidate, PROCESS_HEAP_ENTRY

1.241 InitAtomTable

The **InitAtomTable** function initializes the local atom table and sets the number of hash buckets to the specified size.

```
InitAtomTable: procedure
(
    nSize:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__InitAtomTable@4" );
```

Parameters

nSize

[in] Specifies the number of hash buckets to use for the atom table. If this parameter is zero, the default number of hash buckets are created.

To achieve better performance, specify a prime number in *nSize*.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application need not use this function to use a local atom table. The default number of hash buckets used is 37. If an application does use **InitAtomTable**, however, it should call the function before any other atom-management function.

If an application uses a large number of local atoms, it can reduce the time required to add an atom to the local atom table or to find an atom in the table by increasing the size of the table. However, this increases the amount of memory

required to maintain the table.

The number of buckets in the global atom table cannot be changed. If the atom table has already been initialized, either explicitly by a prior call to **InitAtomTable**, or implicitly by the use of any atom-management function, **InitAtomTable** returns success without changing the number of hash buckets.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Atoms Overview, Atom Functions, AddAtom, DeleteAtom, FindAtom, GetAtomName, GlobalAddAtom, GlobalDeleteAtom, GlobalFindAtom, GlobalGetAtomName

---

1.242 InitializeCriticalSection

The **InitializeCriticalSection** function initializes a critical section object.

```
InitializeCriticalSection: procedure
(
    var lpCriticalSection:  CRITICAL_SECTION
);
    stdcall;
    returns( "eax" );
    external( "__imp__InitializeCriticalSection@4" );
```

Parameters

lpCriticalSection

[out] Pointer to the critical section object.

Return Values

This function does not return a value.

In low memory situations, **InitializeCriticalSection** can raise a STATUS_NO_MEMORY exception.

Remarks

The threads of a single process can use a critical section object for mutual-exclusion synchronization. There is no guarantee about the order in which threads will obtain ownership of the critical section, however, the system will be fair to all threads.

The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function to initialize the object.

After a critical section object has been initialized, the threads of the process can specify the object in the **EnterCriticalSection**, **TryEnterCriticalSection**, or **LeaveCriticalSection** function to provide mutually exclusive access to a shared resource. For similar synchronization between the threads of different processes, use a mutex object.

A critical section object cannot be moved or copied. The process must also not modify the object, but must treat it as logically opaque. Use only the critical section functions provided by the Win32 API to manage critical section objects.

Example

For an example that uses **InitializeCriticalSection**, see Using Critical Section Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateMutex, DeleteCriticalSection, EnterCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection, TryEnterCriticalSection

---

1.243 InitializeCriticalSectionAndSpinCount

The **InitializeCriticalSectionAndSpinCount** function initializes a critical section object and sets the spin count for the critical section.

```
InitializeCriticalSectionAndSpinCount: procedure
(
    var lpCriticalSection:  CRITICAL_SECTION;
        dwSpinCount:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__InitializeCriticalSectionAndSpinCount@8" );
```

Parameters

lpCriticalSection

[in/out] Pointer to the critical section object.

dwSpinCount

[in] Specifies the spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin *dwSpinCount* times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Windows 2000: If the high-order bit is set, the function preallocates the event used by the **EnterCriticalSection** function. Do not set this bit if you are creating a large number of critical section objects, because it will consume a significant amount of nonpaged pool.

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**.

In low memory situations, **InitializeCriticalSectionAndSpinCount** can raise a STATUS_NO_MEMORY exception.

Remarks

The threads of a single process can use a critical section object for mutual-exclusion synchronization. There is no guarantee about the order in which threads will obtain ownership of the critical section, however, the system will be fair to all threads.

The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function to initialize the object. You can subsequently modify the spin count by calling the **SetCriticalSectionSpinCount** function.

After a critical section object has been initialized, the threads of the process can specify the object in the **EnterCriticalSection, TryEnterCriticalSection,** or **LeaveCriticalSection** function to provide mutually exclusive access to a shared resource. For similar synchronization between the threads of different processes, use a mutex object.

A critical section object cannot be moved or copied. The process must also not modify the object, but must treat it as logically opaque. Use only the critical section functions provided by the Win32 API to manage critical section objects.

The spin count is useful for critical sections of short duration that can experience high levels of contention. Consider a worst-case scenario, in which an application on an SMP system has two or three threads constantly allocating and releasing memory from the heap. The application serializes the heap with a critical section. In the worst-case scenario, contention for the critical section is constant, and each thread makes an expensive call to the **WaitForSingleObject** function. However, if the spin count is set properly, the calling thread will not immediately call **WaitForSingleObject** when contention occurs. Instead, the calling thread can acquire ownership of the critical section if it is released during the spin operation.

You can improve performance significantly by choosing a small spin count for a critical section of short duration. The heap manager uses a spin count of roughly 4000 for its per-heap critical sections. This gives great performance and scalability in almost all worst-case scenarios.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, InitializeCriticalSection, SetCriticalSectionSpinCount, WaitForSingleObject

---

1.244 InterlockedCompareExchange

The **InterlockedCompareExchange** function performs an atomic comparison of the specified values and exchanges the values, based on the outcome of the comparison. The function prevents more than one thread from using the same variable simultaneously.

If you are exchanging pointer values, this function has been superseded by the **`InterlockedCompareExchangePointer`** function.

```
InterlockedCompareExchange: procedure
(
    var Destination:    LONG;
        Exchange:       LONG;
        Comperand:      LONG
);
    stdcall;
    returns( "eax" );
    external( "__imp__InterlockedCompareExchange@12" );
```

Parameters

Destination

[in/out] Specifies the address of the destination value. The sign is ignored.

Exchange

[in] Specifies the exchange value. The sign is ignored.

Comperand

[in] Specifies the value to compare to *Destination*. The sign is ignored.

Return Values

The return value is the initial value of the destination.

Remarks

The functions **InterlockedCompareExchange**, InterlockedDecrement, InterlockedExchange, InterlockedExchange-Add, and InterlockedIncrement provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The **InterlockedCompareExchange** function performs an atomic comparison of the *Destination* value with the *Comperand* value. If the *Destination* value is equal to the *Comperand* value, the *Exchange* value is stored in the address specified by *Destination*. Otherwise, no operation is performed.

The variables for **InterlockedCompareExchange** must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor *x*86 systems and any non-x86 systems.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, Interlocked Variable Access, InterlockedCompareExchange-Pointer, InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd, InterlockedIncrement

---

1.245 InterlockedDecrement

The **InterlockedDecrement** function decrements (decreases by one) the value of the specified variable and checks the resulting value. The function prevents more than one thread from using the same variable simultaneously.

```
InterlockedDecrement: procedure
(
    var lpAddend:   LONG
);
    stdcall;
    returns( "eax" );
    external( "__imp__InterlockedDecrement@4" );
```

Parameters

lpAddend

[in/out] Pointer to the variable to decrement.

Return Values

Windows 98, Windows NT 4.0 and later: The return value is the resulting decremented value.

Windows 95, Windows NT 3.51 and earlier: If the result of the operation is zero, the return value is zero. If the result of the operation is less than zero, the return value is negative, but it is not necessarily equal to the result. If the result of the operation is greater than zero, the return value is positive, but it is not necessarily equal to the result.

Remarks

The functions **InterlockedDecrement**, **InterlockedCompareExchange**, **InterlockedExchange**, **InterlockedEx-**

changeAdd, and **InterlockedIncrement** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The variable pointed to by the *lpAddend* parameter must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor *x*86 systems and any non-x86 systems.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, Interlocked Variable Access, InterlockedCompareExchange, InterlockedExchange, InterlockedExchangeAdd, InterlockedIncrement

---

1.246 InterlockedExchange

The **InterlockedExchange** function atomically exchanges a pair of values. The function prevents more than one thread from using the same variable simultaneously.

If you are exchanging pointer values, this function has been superseded by the **`InterlockedExchangePointer`** function.

```
InterlockedExchange: procedure
(
    var Target: LONG;
        Value:  LONG
);
    stdcall;
    returns( "eax" );
    external( "__imp__InterlockedExchange@8" );
```

Parameters

Target

[in/out] Pointer to the value to exchange. The function sets this variable to *Value*, and returns its prior value.

Value

[in] Specifies a new value for the variable pointed to by *Target*.

Return Values

The function returns the initial value pointed to by *Target*.

Remarks

The functions **InterlockedExchange**, **InterlockedCompareExchange**, **InterlockedDecrement**, **InterlockedExchangeAdd**, and **InterlockedIncrement** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The variable pointed to by the *Target* parameter must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor *x*86 systems and any non-x86 systems.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, Interlocked Variable Access, InterlockedCompareExchange, InterlockedDecrement, InterlockedExchangeAdd, InterlockedExchangePointer, InterlockedIncrement

---

1.247 InterlockedExchangeAdd

The **InterlockExchangeAdd** function performs an atomic addition of an increment value to an addend variable. The function prevents more than one thread from using the same variable simultaneously.

```
InterlockedExchangeAdd: procedure
(
    var Addend:     LONG;
        Increment:  LONG
);
    stdcall;
    returns( "eax" );
    external( "__imp__InterlockedExchangeAdd@8" );
```

Parameters

Addend

[in/out] Pointer to the number that will have the *Increment* number added to it.

Increment

[in] Specifies the number to be added to the variable pointed to by the *Addend* parameter.

Return Values

The return value is the initial value of the variable pointed to by the *Addend* parameter.

Remarks

The functions **InterlockedExchangeAdd**, **InterlockedCompareExchange, InterlockedDecrement, InterlockedExchange**, and **InterlockedIncrement** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The **InterlockedExchangeAdd** function performs an atomic addition of the *Increment* value to the value pointed to by *Addend*. The result is stored in the address specified by *Addend*. The initial value of the variable pointed to by *Addend* is returned as the function value.

The variables for **InterlockedExchangeAdd** must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor *x*86 systems and any non-x86 systems.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, Interlocked Variable Access, InterlockedCompareExchange, InterlockedDecrement, InterlockedExchange, InterlockedIncrement

---

1.248 InterlockedIncrement

The **InterlockedIncrement** function increments (increases by one) the value of the specified variable and checks the resulting value. The function prevents more than one thread from using the same variable simultaneously.

```
InterlockedIncrement: procedure
(
    var lpAddend:   LONG
);
    stdcall;
    returns( "eax" );
    external( "__imp__InterlockedIncrement@4" );
```

Parameters

lpAddend

[in/out] Pointer to the variable to increment.

Return Values

Windows 98, Windows NT 4.0 and later: The return value is the resulting incremented value.

Windows 95, Windows NT 3.51 and earlier: If the result of the operation is zero, the return value is zero. If the result of the operation is less than zero, the return value is negative, but it is not necessarily equal to the result. If the result of the operation is greater than zero, the return value is positive, but it is not necessarily equal to the result.

Remarks

The functions **InterlockedIncrement, InterlockedCompareExchange, InterlockedDecrement**, **InterlockedExchange**, and **InterlockedExchangeAdd** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The variable pointed to by the *lpAddend* parameter must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor *x*86 systems and any non-x86 systems.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, Interlocked Variable Access, InterlockedCompareExchange, InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd

---

1.249 IsBadCodePtr

The **IsBadCodePtr** function determines whether the calling process has read access to the memory at the specified address.

```
IsBadCodePtr: procedure
(
    lpfn:   procedure
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsBadCodePtr@4" );
```

Parameters

lpfn

[in] Pointer to an address in memory.

Return Values

If the calling process has read access to the specified memory, the return value is zero.

If the calling process does not have read access to the specified memory, the return value is nonzero. To get extended error information, call **GetLastError**.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value This behavior is by design, as a debugging aid.

Remarks

IsBadCodePtr checks the read access only at the specified address and does not guarantee read access to a range of memory.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr

---

1.250 IsBadReadPtr

The **IsBadReadPtr** function verifies that the calling process has read access to the specified range of memory.

```
IsBadReadPtr: procedure
(
    var lp:     var;
    var ucb:    uns32
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsBadReadPtr@8" );
```

Parameters

lp

[in] Pointer to the first byte of the memory block.

ucb

[in] Specifies the size, in bytes, of the memory block. If this parameter is zero, the return value is zero.

Return Values

If the calling process has read access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have read access to all bytes in the specified memory range, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has read access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, IsBadCodePtr, IsBadStringPtr, IsBadWritePtr

---

1.251 IsBadStringPtr

The **IsBadStringPtr** function verifies that the calling process has read access to a range of memory pointed to by a string pointer.

```
IsBadStringPtr: procedure
(
    lpsz:       string;
    ucchMax:    dword
```

```
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsBadStringPtrA@8" );
```

Parameters

lpsz

[in] Pointer to a null-terminated string, either Unicode or ASCII.

ucchMax

[in] Specifies the maximum size, in **TCHARs**, of the string. The function checks for read access in all bytes up to the string's terminating null character or up to the number of bytes specified by this parameter, whichever is smaller. If this parameter is zero, the return value is zero.

Return Values

If the calling process has read access to all characters up to the string's terminating null character or up to the number of characters specified by *ucchMax*, the return value is zero.

If the calling process does not have read access to all characters up to the string's terminating null character or up to the number of characters specified by *ucchMax*, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to the entire memory range specified, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has read access to some, but not all, of the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hff
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, IsBadCodePtr, IsBadReadPtr, IsBadWritePtr

1.252 IsBadWritePtr

The **IsBadWritePtr** function verifies that the calling process has write access to the specified range of memory.

```
IsBadWritePtr: procedure
(
    var lp:     var;
        ucb:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsBadWritePtr@8" );
```

Parameters

lp

[in] Pointer to the first byte of the memory block.

ucb

[in] Specifies the size, in bytes, of the memory block. If this parameter is zero, the return value is zero.

Return Values

If the calling process has write access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have write access to all bytes in the specified memory range, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have write access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has write access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has write access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

IsBadWritePtr is not multithread safe. To use it properly on a pointer shared by multiple threads, call it inside a critical region of code that allows only one thread to access the memory being checked. Use operating system–level objects such as critical sections or mutexes or the interlocked functions to create the critical region of code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, IsBadCodePtr, IsBadHugeReadPtr, IsBadHugeWritePtr, IsBadReadPtr

---

1.253 IsDBCSLeadByte

The **IsDBCSLeadByte** function determines whether a specified byte is a lead byte—that is, the first byte of a character in a double-byte character set (DBCS).

```
IsDBCSLeadByte: procedure
(
    TestChar:   byte
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsDBCSLeadByte@4" );
```

Parameters

TestChar

[in] Specifies the byte to be tested.

Return Values

If the byte is a lead byte, it returns a nonzero value.

If the byte is not a lead byte, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Lead bytes are unique to double-byte character sets. A lead byte introduces a double-byte character. Lead bytes occupy a specific range of byte values. The **IsDBCSLeadByte** function uses the ANSI code page to check lead-byte ranges. To specify a different code page, use the **IsDBCSLeadByteEx** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Functions, IsDBCSLeadByteEx, MultiByteToWideChar

---

1.254 IsDBCSLeadByteEx

The **IsDBCSLeadByteEx** function determines whether a specified byte is a lead byte that is, the first byte of a character in a double-byte character set (DBCS).

```
IsDBCSLeadByteEx: procedure
(
    CodePage:   dword;
    TestChar:   byte
);
```

```
    stdcall;
    returns( "eax" );
    external( "__imp__IsDBCSLeadByteEx@8" );
```

Parameters

CodePage

[in] Identifier of the code page to use to check lead-byte ranges. Can be one of the values given in the "Code-Page Identifiers" table in Unicode and Character Set Constants or one of the following predefined values.

Value	Meaning
0	Use system default ANSI code page.
CP_ACP	Use system default ANSI code page.
CP_OEMCP	Use system default OEM code page.

TestChar

[in] Byte to test.

Return Values

If the byte is a lead byte, it returns a nonzero value.

If the byte is not a lead byte, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Functions

---

1.255 IsDebuggerPresent

The **IsDebuggerPresent** function determines whether the calling process is running under the context of a debugger.

```
IsDebuggerPresent: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__IsDebuggerPresent@0" );
```

Parameters

This function has no parameters.

Return Value

If the current process is running in the context of a debugger, the return value is nonzero.

If the current process is not running in the context of a debugger, the return value is zero.

Remarks

This function allows an application to determine whether or not it is being debugged, so that it can modify its behavior. For example, an application could provide additional information using the **OutputDebugString** function if it is being debugged.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, OutputDebugString

1.256 IsProcessorFeaturePresent

The **IsProcessorFeaturePresent** function determines whether the specified processor feature is supported by the current computer.

```
IsProcessorFeaturePresent: procedure
(
    ProcessorFeature:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsProcessorFeaturePresent@4" );
```

Parameters

ProcessorFeature

[in] Specifies the processor feature to be tested. This parameter can be one of the following values.

Value	Meaning
PF_FLOATING_POINT_PRECISION_ERRATA	In rare circumstances, a floating-point precision error can occur (Pentium).
PF_FLOATING_POINT_EMULATED	Floating-point operations are emulated using a software emulator. **Windows 2000:** This function returns a nonzero value if floating-point operations are emulated; otherwise, it returns zero. **Windows NT 4.0:** This function returns zero if floating-point operations are emulated; otherwise, it returns a nonzero value. This behavior is a bug that is fixed in later versions.
PF_COMPARE_EXCHANGE_DOUBLE	The compare and exchange double operation is available (Pentium, MIPS, and Alpha).
PF_MMX_INSTRUCTIONS_AVAILABLE	The MMX instruction set is available.
PF_XMMI_INSTRUCTIONS_AVAILABLE	The XMMI instruction set is available.

PF_3DNOW_INSTRUCTIONS_AVAILABLE	The 3D-Now instruction set is available.
PF_RDTSC_INSTRUCTION_AVAILABLE	The RDTSC instruction is available.
PF_PAE_ENABLED	The processor is PAE-enabled.

Return Values

If feature is supported, the return value is a nonzero value.

If the feature is not supported, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions

1.257 IsValidCodePage

The **IsValidCodePage** determines whether a specified code page is valid.

```
IsValidCodePage: procedure
(
    CodePage:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsValidCodePage@4" );
```

Parameters

CodePage

[in] Specifies the code page to check. Each code page is identified by a unique number.

Return Values

If the code page is valid, the return values is a nonzero value.

If the code page is not valid, the return value is zero.

Remarks

A code page is considered valid only if it is installed in the system.

The following are the code-page identifiers.

Identifier	Meaning
037	EBCDIC
437	MS-DOS United States
500	EBCDIC "500V1"

708	Arabic (ASMO 708)
709	Arabic (ASMO 449+, BCON V4)
710	Arabic (Transparent Arabic)
720	Arabic (Transparent ASMO)
737	Greek (formerly 437G)
775	Baltic
850	MS-DOS Multilingual (Latin I)
852	MS-DOS Slavic (Latin II)
855	IBM Cyrillic (primarily Russian)
857	IBM Turkish
860	MS-DOS Portuguese
861	MS-DOS Icelandic
862	Hebrew
863	MS-DOS Canadian-French
864	Arabic
865	MS-DOS Nordic
866	MS-DOS Russian
869	IBM Modern Greek
874	Thai
875	EBCDIC
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1026	EBCDIC
1250	Windows 3.1 Eastern European
1251	Windows 3.1 Cyrillic
1252	Windows 3.1 US (ANSI)
1253	Windows 3.1 Greek
1254	Windows 3.1 Turkish
1255	Hebrew
1256	Arabic
1257	Baltic

1361	Korean (Johab)
10000	Macintosh Roman
10001	Macintosh Japanese
10006	Macintosh Greek I
10007	Macintosh Cyrillic
10029	Macintosh Latin 2
10079	Macintosh Icelandic
10081	Macintosh Turkish

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetACP, GetCPInfo, GetOEMCP

1.258 IsValidLocale

The **IsValidLocale** function determines whether a specified locale identifier is valid. Currently, the function tests whether a locale identifier is installed or supported on the calling system, based on the specified validity test.

```
IsValidLocale: procedure
(
    Locale:     LCID;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__IsValidLocale@8" );
```

Parameters

Locale

[in] Specifies the locale identifier to be validated. You can use the **MAKELCID** macro to create a locale identifier.

dwFlags

[in] Specifies the validity test to apply to the locale identifier. This parameter can be one of the following values.

Value	Meaning
LCID_INSTALLED	Test whether the locale identifier is both supported and installed.
LCID_SUPPORTED	Test whether the locale identifier is supported.

Return Values

If the locale identifier passes the specified validity test, the return values is a nonzero value.

If the locale identifier does not pass the specified validity test, the return value is zero.

Remarks

If the LCID_INSTALLED flag is specified and this function returns a nonzero value, the locale identifier is both supported and installed on the system. Having an LCID installed implies that the full level of language support is available for this locale. This includes code page translation tables, keyboard layouts, fonts, sorting and locale data.

If LCID_SUPPORTED is specified and this function returns zero, the locale identifier is supported in the release, but not necessarily installed on the system.

For more information, see Locales.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetLocaleInfo

---

1.259 LCMapString

The **LCMapString** function either maps an input character string to another using a specified transformation or generates a sort key for the input string.

```
LCMapString: procedure
(
        Locale:     LCID;
        dwMapFlags: dword;
        lpSrcStr:   string;
        cchSrc:     dword;
    var lpDestStr:  var;
        cchDest:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LCMapStringA@24" );
```

Parameters

Locale

[in] Specifies a locale identifier. The locale provides a context for the string mapping or sort key generation. An application can use the **MAKELCID** macro to create a locale identifier.

dwMapFlags

[in] Specifies the type of transformation used during string mapping or the type of sort key generated. An application can specify one or more of the following options. Restrictions are noted following the table.

Option	Meaning
LCMAP_BYTEREV	**Windows NT/2000:** Use byte reversal. For example, if you pass in 0x3450 0x4822 the result is 0x5034 0x2248.
LCMAP_FULLWIDTH	Uses wide characters (where applicable).

LCMAP_HALFWIDTH	Uses narrow characters (where applicable).
LCMAP_HIRAGANA	Hiragana.
LCMAP_KATAKANA	Katakana.
LCMAP_LINGUISTIC_CASING	Uses linguistic rules for casing, rather than file system rules (the default). Valid with LCMAP_LOWERCASE or LCMAP_UPPERCASE only.
LCMAP_LOWERCASE	Uses lowercase.
LCMAP_SIMPLIFIED_CHINESE	**Windows NT 4.0 and later:** Maps traditional Chinese characters to simplified Chinese characters.
LCMAP_SORTKEY	Produces a normalized wide character–sort key. For more information, see *lpDestStr* and Remarks.
LCMAP_TRADITIONAL_CHINESE	**Windows NT 4.0 and later:** Maps simplified Chinese characters to traditional Chinese characters.
LCMAP_UPPERCASE	Uses uppercase.

The following flags are used only with the LCMAP_SORTKEY flag.

Flag	Meaning
NORM_IGNORECASE	Ignores case.
NORM_IGNOREKANATYPE	Does not differentiate between Hiragana and Katakana characters. Corresponding Hiragana and Katakana will compare as equal.
NORM_IGNORENONSPACE	Ignores nonspacing. This flag also removes Japanese accent characters.
NORM_IGNORESYMBOLS	Ignores symbols.
NORM_IGNOREWIDTH	Does not differentiate between a single-byte character and the same character as a double-byte character.
SORT_STRINGSORT	Treats punctuation the same as symbols.

If the LCMAP_SORTKEY flag *is not* specified, the **LCMapString** function performs string mapping. In this case the following restrictions apply:

LCMAP_LOWERCASE and LCMAP_UPPERCASE are mutually exclusive.

LCMAP_HIRAGANA and LCMAP_KATAKANA are mutually exclusive.

LCMAP_HALFWIDTH and LCMAP_FULLWIDTH are mutually exclusive.

LCMAP_TRADITIONAL_CHINESE and LCMAP_SIMPLIFIED_CHINESE are mutually exclusive.

LCMAP_LOWERCASE and LCMAP_UPPERCASE are not valid in combination with any of these flags: LCMAP_HIRAGANA, LCMAP_KATAKANA, LCMAP_HALFWIDTH, LCMAP_FULLWIDTH.

When the LCMAP_SORTKEY flag *is* specified, the **LCMapString** function generates a sort key. In this case the following restriction applies:

LCMAP_SORTKEY is mutually exclusive with all other LCMAP_* flags, with the sole exception of LCMAP_BYTEREV.

lpSrcStr

[in] Pointer to a source string that the function maps or uses for sort key generation.

cchSrc

[in] Specifies the number of **TCHARs** in the string pointed to by the *lpSrcStr* parameter.

This count can include the NULL terminator, or not include it. If the NULL terminator is included in the character count, it does not greatly affect the mapping behavior. That is because NULL is considered to be unsortable, and always maps to itself.

A *cchSrc* value of –1 specifies that the string pointed to by *lpSrcStr* is null-terminated. If this is the case, and **LCMapString** is being used in its string-mapping mode, the function calculates the string's length itself, and null-terminates the mapped string stored into **lpDestStr*.

lpDestStr

[out] Pointer to a buffer that receives the mapped string or sort key.

If LCMAP_SORTKEY is specified, **LCMapString** stores a sort key into the buffer. The sort key is stored as an array of byte values in the following format:

```
[all Unicode sort weights] 0x01 [all Diacritic weights] 0x01 [all Case
weights] 0x01 [all Special weights] 0x00
```

Note that the sort key is null-terminated. This is true regardless of the value of *cchSrc*. Also note that, even if some of the sort weights are absent from the sort key, due to the presence of one or more ignore flags in *dwMapFlags*, the 0x01 separators and the 0x00 terminator are still present.

cchDest

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by *lpDestStr*.

If the function is being used for string mapping, the size is a character count. If space for a NULL terminator is included in *cchSrc*, then *cchDest* must also include space for a NULL terminator.

If the function is being used to generate a sort key, the size is a byte count. This byte count must include space for the sort key 0x00 terminator.

If *cchDest* is zero, the function's return value is the number of characters, or bytes if LCMAP_SORTKEY is specified, required to hold the mapped string or sort key. In this case, the buffer pointed to by *lpDestStr* is not used.

Return Values

If the function succeeds, and the value of *cchDest* is nonzero, the return value is the number of characters, or bytes if LCMAP_SORTKEY is specified, written to the buffer. This count includes room for a NULL terminator.

If the function succeeds, and the value of *cchDest* is zero, the return value is the size of the buffer in characters, or bytes if LCMAP_SORTKEY is specified, required to receive the translated string or sort key. This size includes room for a NULL terminator.

If the function fails, the return value is 0. 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

Remarks

The mapped string is null terminated if the source string is null terminated.

The ANSI version of this function maps strings to and from Unicode based on the specified LCID's default ANSI code page.

For the ANSI version of this function, the LCMAP_UPPERCASE flag produces the same result as **AnsiUpper** in the locale. Likewise, the LCMAP_LOWERCASE flag produces the same result as **AnsiLower**. This function always maps a single character to a single character.

If LCMAP_UPPERCASE or LCMAP_LOWERCASE is set and if LCMAP_SORTKEY is not set, the *lpSrcStr* and *lpDestStr* pointers can be the same. Otherwise, the *lpSrcStr* and *lpDestStr* pointers must not be the same. If they are the same, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

If the LCMAP_HIRAGANA flag is specified to map Katakana characters to Hiragana characters, and LCMAP_FULLWIDTH is not specified, the function only maps full-width characters to Hiragana. In this case, any half-width Katakana characters are placed as-is in the output string, with no mapping to Hiragana. An application must specify LCMAP_FULLWIDTH if it wants half-width Katakana characters mapped to Hiragana.

Even if the Unicode version of this function is called, the output string is only in **WCHAR** or **CHAR** format if the string mapping mode of **LCMapString** is used. If the sort key generation mode is used, specified by LCMAP_SORTKEY, the output is an array of byte values. To compare sort keys, use a byte-by-byte comparison.

An application can call the function with the NORM_IGNORENONSPACE and NORM_IGNORESYMBOLS flags set, and all other options flags cleared, in order to simply strip characters from the input string. If this is done with an input string that is not null-terminated, it is possible for **LCMapString** to return an empty string and not return an error.

The **LCMapString** function ignores the Arabic Kashida. If an application calls the function to create a sort key for a string containing an Arabic Kashida, there will be no sort key value for the Kashida.

The function treats the hyphen and apostrophe a bit differently than other punctuation symbols, so that words like coop and co-op stay together in a list. All punctuation symbols other than the hyphen and apostrophe sort before the alphanumeric characters. An application can change this behavior by setting the SORT_STRINGSORT flag. See **`CompareString`** for a more detailed discussion of this issue.

When **LCMapString** is used to generate a sort key, by setting the LCMAP_SORTKEY flag, the sort key stored into **lpDestStr* may contain an odd number of bytes. The LCMAP_BYTEREV option only reverses an even number of bytes. If both options are chosen, the last (odd-positioned) byte in the sort key is not reversed. If the terminating 0x00 byte is an odd-positioned byte, then it remains the last byte in the sort key. If the terminating 0x00 byte is an even-positioned byte, it exchanges positions with the byte that precedes it.

When LCMAP_SORTKEY flag is specified, the function generates a sort key that, when used in **strcmp**, produces the same order as when the original string is used in **CompareString**. When LCMAP_SORTKEY flag is specified, the output string is a string, but the character values are not meaningful display values.

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only locale. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, AnsiLower, AnsiUpper, CompareString, FoldString, MAKELCID

1.260 LeaveCriticalSection

The **LeaveCriticalSection** function releases ownership of the specified critical section object.

```
LeaveCriticalSection: procedure
(
    var lpCriticalSection:  CRITICAL_SECTION
);
    stdcall;
    returns( "eax" );
    external( "__imp__LeaveCriticalSection@4" );
```

Parameters

lpCriticalSection

[in/out] Pointer to the critical section object.

Return Values

This function does not return a value.

Remarks

The threads of a single process can use a critical-section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical-section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call the **`InitializeCriticalSection`** or **`InitializeCriticalSectionAndSpinCount`** function to initialize the object.

A thread uses the **`EnterCriticalSection`** or **`TryEnterCriticalSection`** function to acquire ownership of a critical section object. To release its ownership, the thread must call **LeaveCriticalSection** once for each time that it entered the critical section.

If a thread calls **LeaveCriticalSection** when it does not have ownership of the specified critical section object, an error occurs that may cause another thread using **EnterCriticalSection** to wait indefinitely.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

Example

For an example that uses **LeaveCriticalSection**, see Using Critical Section Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, DeleteCriticalSection, EnterCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, TryEnterCriticalSection

---

1.261 LoadLibrary

The **LoadLibrary** function maps the specified executable module into the address space of the calling process.

For additional load options, use the **`LoadLibraryEx`** function.

```
LoadLibrary: procedure
(
    lpFileName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__LoadLibraryA@4" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that names the executable module (either a .dll or .exe file). The name

specified is the file name of the module and is not related to the name stored in the library module itself, as specified by the **LIBRARY** keyword in the module-definition (.def) file.

If the string specifies a path but the file does not exist in the specified directory, the function fails. When specifying a path, be sure to use backslashes (\), not forward slashes (/).

If the string does not specify a path, the function uses a standard search strategy to find the file. See the **Remarks** for more information.

Return Values

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

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

Windows 95: If you are using **LoadLibrary** to load a module that contains a resource whose numeric identifier is greater than 0x7FFF, **LoadLibrary** fails. If you are attempting to load a 16-bit DLL directly from 32-bit code, **LoadLibrary** fails. If you are attempting to load a DLL whose subsystem version is greater than 4.0, **LoadLibrary** fails. If your **DllMain** function tries to call the Unicode version of a Win32 function, **LoadLibrary** fails.

Remarks

LoadLibrary can be used to map a DLL module and return a handle that can be used in **GetProcAddress** to get the address of a DLL function. **LoadLibrary** can also be used to map other executable modules. For example, the function can specify an .exe file to get a handle that can be used in **FindResource** or **LoadResource**. However, do not use **LoadLibrary** to run an .exe file, use the **CreateProcess** function.

If the module is a DLL not already mapped for the calling process, the system calls the DLL's **DllMain** function with the DLL_PROCESS_ATTACH value. If the DLL's entry-point function does not return TRUE, **LoadLibrary** fails and returns NULL. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.)

It is not safe to call **LoadLibrary** from **DllMain**. For more information, see the Remarks section in **DllMain**.

Module handles are not global or inheritable. A call to **LoadLibrary** by one process does not produce a handle that another process can use — for example, in calling **GetProcAddress**. The other process must make its own call to **LoadLibrary** for the module before calling **GetProcAddress**.

If no file name extension is specified in the *lpFileName* parameter, the default library extension .dll is appended. However, the file name string can include a trailing point character (.) to indicate that the module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file in the following sequence:

The directory from which the application loaded.

The current directory.

Windows 95/98: The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

Windows NT/ 2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/ 2000: The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.

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

The directories that are listed in the PATH environment variable.

The first directory searched is the one directory containing the image file used to create the calling process (for more information, see the **CreateProcess** function). Doing this allows private dynamic-link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable.

Windows 2000: If a path is specified and there is a redirection file for the application, the function searches for the module in the application's directory. If the module exists in the application's directory, the **LoadLibrary** function ignores the specified path and loads the module from the application's directory. If the module does not exist in the application's directory, **LoadLibrary** loads the module from the specified directory.

The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: **_declspec(thread)**. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary** or **LoadLibraryEx**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of **_declspec(thread)**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, DllMain, FindResource, FreeLibrary, GetProcAddress, GetSystemDirectory, GetWindowsDirectory, LoadLibraryEx, LoadResource

---

1.262 LoadLibraryEx

The **LoadLibraryEx** function maps the specified executable module into the address space of the calling process. The executable module can be a .dll or an .exe file. The specified module may cause other modules to be mapped into the address space.

```
LoadLibraryEx: procedure
(
    lpFileName: string;
    hFile:      dword;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LoadLibraryExA@12" );
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that names the executable module (either a .dll or an .exe file). The name specified is the file name of the executable module. This name is not related to the name stored in a library module itself, as specified by the **LIBRARY** keyword in the module-definition (.DEF) file.

If the string specifies a path, but the file does not exist in the specified directory, the function fails. When specifying a path, be sure to use backslashes (\), not forward slashes (/).

If the string does not specify a path, and the file name extension is omitted, the function appends the default library extension .dll to the file name. However, the file name string can include a trailing point character (.) to indicate that the module name has no extension.

If the string does not specify a path, the function uses a standard search strategy to find the file. See the **Remarks** for more information.

If mapping the specified module into the address space causes the system to map in other, associated executable modules, the function can use either the standard search strategy or an alternate search strategy to find those modules. See the **Remarks** for more information.

hFile

This parameter is reserved for future use. It must be NULL.

dwFlags

[in] Specifies the action to take when loading the module. If no flags are specified, the behavior of this function is identical to that of the **LoadLibrary** function. This parameter can be one of the following values.

Flag	Meaning
DONT_RESOLVE_DLL_REFERENCES	**Windows NT/ 2000:** If this value is used, and the executable module is a DLL, the system does not call **DllMain** for process and thread initialization and termination. Also, the system does not load additional executable modules that are referenced by the specified module. If this value is not used, and the executable module is a DLL, the system calls **DllMain** for process and thread initialization and termination. The system loads additional executable modules that are referenced by the specified module.
LOAD_LIBRARY_AS_DATAFILE	If this value is used, the system maps the file into the calling process's virtual address space as if it were a data file. Nothing is done to execute or prepare to execute the mapped file. Use this flag when you want to load a DLL only to extract messages or resources from it. **Windows NT/ 2000:** You can use the resulting module handle with any Win32 functions that operate on resources. **Windows 95/98:** You can use the resulting module handle only with resource management functions such as **EnumResourceLanguages, EnumResourceNames, EnumResourceTypes, FindResource, FindResourceEx, LoadResource,** and **SizeofResource**. You cannot use this handle with specialized resource management functions such as **LoadBitmap, LoadCursor, LoadIcon, LoadImage,** and **LoadMenu**.
LOAD_WITH_ALTERED_SEARCH_PATH	If this value is used, and *lpFileName* specifies a path, the system uses the alternate file search strategy discussed in the Remarks section to find associated executable modules that the specified module causes to be loaded. If this value is not used, or if *lpFileName* does not specify a path, the system uses the standard search strategy discussed in the Remarks section to find associated executable modules that the specified module causes to be loaded.

Return Values

If the function succeeds, the return value is a handle to the mapped executable module.

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

Windows 95: If you are using **LoadLibraryEx** to load a module that contains a resource whose numeric identifier is greater than 0x7FFF, **LoadLibraryEx** fails. If you are attempting to load a 16-bit DLL directly from 32-bit code, **LoadLibraryEx** fails. If you are attempting to load a DLL whose subsystem version is greater than 4.0, **LoadLibraryEx** fails. If your **DllMain** function tries to call the Unicode version of a Win32 function, **LoadLibraryEx** fails.

Remarks

The calling process can use the handle returned by this function to identify the module in calls to the **GetProcAddress, FindResource,** and **LoadResource** functions.

The **LoadLibraryEx** function is very similar to the **LoadLibrary** function. The differences consist of a set of optional behaviors that **LoadLibraryEx** provides. First, **LoadLibraryEx** can map a DLL module without calling the **DllMain** function of the DLL. Second, **LoadLibraryEx** can use either of two file search strategies to find executable modules that are associated with the specified module. Third, **LoadLibraryEx** can load a module in a way that is

optimized for the case where the module will never be executed, loading the module as if it were a data file. You select these optional behaviors by setting the *dwFlags* parameter; if *dwFlags* is zero, **LoadLibraryEx** behaves identically to **LoadLibrary.**

It is not safe to call **LoadLibraryEx** from **DllMain**. For more information, see the Remarks section in **DllMain**.

If no path is specified in the *lpFileName* parameter, and the base file name does not match the base file name of a loaded module, the **LoadLibraryEx** function uses the same standard file search strategy that **LoadLibrary, SearchPath,** and **OpenFile** use to find the executable module and any associated executable modules that it causes to be loaded. This standard strategy searches for a file in the following sequence:

The directory from which the application loaded.

The current directory.

Windows 95/98: The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

Windows NT/ 2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/ 2000: The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.

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

The directories that are listed in the PATH environment variable.

If a path is specified, and the *dwFlags* parameter is set to LOAD_WITH_ALTERED_SEARCH_PATH, the **LoadLibraryEx** function uses an alternate file search strategy to find any executable modules that the specified module causes to be loaded. This alternate strategy searches for a file in the following sequence:

The directory specified by the *lpFileName* path. In other words, the directory that the specified executable module is in.

The current directory.

Windows 95/98: The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

Windows NT/ 2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/ 2000: The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.

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

The directories that are listed in the PATH environment variable.

Note that the standard file search strategy and the alternate search strategy differ in just one way: the standard strategy starts its search in the calling application's directory, and the alternate strategy starts its search in the directory of the executable module that **LoadLibraryEx** is loading.

If you specify the alternate search strategy, its behavior continues until all associated executable modules have been located. After the system starts processing DLL initialization routines, the system reverts to the standard search strategy.

Windows 2000: If a path is specified and there is a redirection file associated with the application, the **LoadLibraryEx** function searches for the module in the application directory. If the module exists in the application directory, **LoadLibraryEx** ignores the path specification and loads the module from the application directory. If the module does not exist in the application directory, the function loads the module from the specified directory.

Visual C++: The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: **_declspec(thread)**. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary** or **LoadLibraryEx**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of **_declspec(thread)**.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, DllMain, FindResource, FreeLibrary, GetProcAddress, GetSystemDirectory, GetWindowsDirectory, LoadLibrary, LoadResource, OpenFile, SearchPath

---

1.263 LoadModule

The **LoadModule** function loads and executes an application or creates a new instance of an existing application.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the **CreateProcess** function.

```
LoadModule: procedure
(
        lpModuleName:       string;
    var lpParameterBlock:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__LoadModule@8" );
```

Parameters

lpModuleName

[in] Pointer to a null-terminated string that contains the file name of the application to run. When specifying a path, be sure to use backslashes (\), not forward slashes (/). If the *lpModuleName* parameter does not contain a directory path, the system searches for the executable file in this order:

The directory from which the application loaded.

The current directory.

Windows 95/98: The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

Windows NT/ 2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/ 2000: 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.

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

The directories that are listed in the PATH environment variable.

lpParameterBlock

[in] Pointer to an application-defined **LOADPARMS32** structure that defines the new application's parameter block.

The **LOADPARMS32** structure has the following form:

```
typedef struct tagLOADPARMS32 {

    LPSTR lpEnvAddress;  // address of environment strings

    LPSTR lpCmdLine;     // address of command line

    LPSTR lpCmdShow;     // how to show new program
```

```
    DWORD dwReserved;    // must be zero
} LOADPARMS32;
```

Member	Description
lpEnvAddress	Pointer to an array of null-terminated strings that supply the environment strings for the new process. The array has a value of NULL as its last entry. A value of NULL for this parameter causes the new process to start with the same environment as the calling process.
lpCmdLine	Pointer to a Pascal-style string that contains a correctly formed command line. The first byte of the string contains the number of bytes in the string. The remainder of the string contains the command line arguments, excluding the name of the child process. If there are no command line arguments, this parameter must point to a zero length string; it cannot be NULL.
lpCmdShow	Pointer to a structure containing two **WORD** values. The first value must always be set to two. The second value specifies how the application window is to be shown and is used to supply the **wShowWindow** member of the **`STARTUPINFO`** structure to the **`CreateProcess`** function. See the description of the *nCmdShow* parameter of the **`ShowWindow`** function for a list of acceptable values.
dwReserved	This parameter is reserved; it must be zero.

Set all unused members to NULL, except for **lpCmdLine**, which must point to a null-terminated string if it is not used.

Return Values

If the function succeeds, the return value is greater than 31.

If the function fails, the return value is an error value, which may be one of the following values.

Value	Meaning
0	The system is out of memory or resources.
ERROR_BAD_FORMAT	The .exe file is invalid (non-Win32 .exe or error in .exe image).
ERROR_FILE_NOT_FOUND	The specified file was not found.
ERROR_PATH_NOT_FOUND	The specified path was not found.

Remarks

Win32-based applications should use the **`CreateProcess`** function. In the Win32 API, the implementation of the **LoadModule** function calls **CreateProcess**. The following table shows how each parameter for **CreateProcess** is formed.

CreateProcess parameter	Value
lpszApplicationName	*lpModuleName*
lpszCommandLine	lpParameterBlock->lpCmdLine
lpProcessAttributes	NULL
lpThreadAttributes	NULL
bInheritHandles	FALSE

dwCreationFlags	0
lpEnvironment	lpParameterBlock->lpEnvAddress
lpCurrentDirectory	NULL
lpStartupInfo	The structure is initialized to zero. The **cb** member is set to the size of the structure, and the **wShowWindow** member is set to the value of the second word of the **LoadModule** *lpParameter-Block->lpCmdShow* parameter.
lpProcessInformation.**hProcess**	The handle is immediately closed.
lpProcessInformation.**hThread**	The handle is immediately closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Dynamic-Link Libraries Overview, Dynamic-Link Library Functions, CreateProcess, GetSystemDirectory, GetWindowsDirectory, ShowWindow, STARTUPINFO, WinExec

---

1.264 LoadResource

The **LoadResource** function loads the specified resource into global memory.

```
LoadResource: procedure
(
    hModule:    dword;
    hResInfo:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LoadResource@8" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resource. If *hModule* is NULL, the system loads the resource from the module that was used to create the current process.

hResInfo

[in] Handle to the resource to be loaded. This handle is returned by the **FindResource** or **FindResourceEx** function.

Return Values

If the function succeeds, the return value is a handle to the data associated with the resource.

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

Remarks

The return type of **LoadResource** is **HGLOBAL** for backward compatibility, not because the function returns a han-

dle to a global memory block. Do not pass this handle to the **GlobalLock** or **GlobalFree** function. To obtain a pointer to the resource data, call the **LockResource** function.

The **LoadResource** function is used to load an icon in order to copy its data to another application, followed by **FreeResource** when done.

To use a resource immediately, an application should use the following resource-specific functions to find and load the resource in one call.

Function	Action	To remove resource
FormatMessage	Loads and formats a message-table entry	No action needed
LoadAccelerators	Loads an accelerator table	**DestroyAcceleratorTable**
LoadBitmap	Loads a bitmap resource	**DeleteObject**
LoadCursor	Loads a cursor resource	**DestroyCursor**
LoadIcon	Loads an icon resource	**DestroyIcon**
LoadMenu	Loads a menu resource	**DestroyMenu**
LoadString	Loads a string resource	No action needed

For example, an application can use the **LoadIcon** function to load an icon for display on the screen, followed by **DestroyIcon** when done.

The system automatically deletes these resources when the process that created them terminates, however, calling the appropriate function saves memory and decreases the size of the process's working set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, FindResource, FindResourceEx, LoadLibrary, LoadModule, LockResource

---

1.265 LocalAlloc

The **LocalAlloc** function allocates the specified number of bytes from the heap. Win32 memory management does not provide a separate local heap and global heap.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalAlloc: procedure
(
    uFlags: dword;
    uBytes: SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalAlloc@8" );
```

Parameters

uFlags

[in] Specifies how to allocate memory. If zero is specified, the default is the LMEM_FIXED value. Except for the incompatible combinations that are specifically noted, this parameter can be any combination of the following values.

Value	Meaning
LHND	Combines LMEM_MOVEABLE and LMEM_ZEROINIT.
LMEM_FIXED	Allocates fixed memory. The return value is a pointer to the memory object.
LMEM_MOVEABLE	Allocates movable memory. In Win32, memory blocks are never moved in physical memory, but they can be moved within the default heap. The return value is a handle to the memory object. To translate the handle to a pointer, use the **LocalLock** function. This value cannot be combined with LMEM_FIXED.
LMEM_ZEROINIT	Initializes memory contents to zero.
LPTR	Combines LMEM_FIXED and LMEM_ZEROINIT.
NONZEROLHND	Same as LMEM_MOVEABLE.
NONZEROLPTR	Same as LMEM_FIXED.

The following values are obsolete.

Value	Meaning
LMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **LocalDiscard** function to discard a block. This value cannot be combined with LMEM_FIXED.
LMEM_NOCOMPACT	Ignored. This value is provided only for compatibility with 16-bit Windows.
LMEM_NODISCARD	Ignored. This value is provided only for compatibility with 16-bit Windows.

uBytes

[in] Specifies the number of bytes to allocate. If this parameter is zero and the *uFlags* parameter specifies LMEM_MOVEABLE, the function returns a handle to a memory object that is marked as discarded.

Return Values

If the function succeeds, the return value is a handle to the newly allocated memory object.

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

Remarks

If the heap does not contain sufficient free space to satisfy the request, **LocalAlloc** returns NULL. Because NULL is used to indicate an error, virtual address zero is never allocated. It is, therefore, easy to detect the use of a NULL pointer.

If this function succeeds, it allocates at least the amount requested. If the amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual number of bytes allocated, use the **LocalSize** function.

To free the memory, use the **LocalFree** function.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalFree, LocalLock, LocalReAlloc, LocalSize

---

1.266 LocalFileTimeToFileTime

The **LocalFileTimeToFileTime** function converts a local file time to a file time based on the Coordinated Universal Time (UTC).

```
LocalFileTimeToFileTime: procedure
(
    var lpLocalFileTime:    FILETIME;
    var lpFileTime:         FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalFileTimeToFileTime@8" );
```

Parameters

lpLocalFileTime

[in] Pointer to a **FILETIME** structure that specifies the local file time to be converted into a UTC-based file time.

lpFileTime

[out] Pointer to a **FILETIME** structure to receive the converted UTC-based file time. This parameter cannot be the same as the *lpLocalFileTime* parameter.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

LocalFileTimeToFileTime uses the current settings for the time zone and daylight saving time. Therefore, if it is daylight saving time, this function will take daylight saving time into account, even if the time you are converting is in standard time.

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, FileTimeToLocalFileTime

---

1.267 LocalFlags

The **LocalFlags** function returns information about the specified local memory object.

Note This function is provided only for compatibility with 16-bit versions of Windows.

```
LocalFlags: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalFlags@4" );
```

Parameters

hMem

[in] Handle to the local memory object. This handle is returned by either the **LocalAlloc** or **LocalReAlloc** function.

Return Values

If the function succeeds, the return value specifies the allocation values and the lock count for the memory object.

If the function fails, the return value is LMEM_INVALID_HANDLE, indicating that the local handle is not valid. To get extended error information, call **GetLastError**.

Remarks

The low-order byte of the low-order word of the return value contains the lock count of the object. To retrieve the lock count from the return value, use the LMEM_LOCKCOUNT mask with the bitwise AND (&) operator. The lock count of memory objects allocated with LMEM_FIXED is always zero.

The high-order byte of the low-order word of the return value indicates the allocation values of the memory object. It can be zero or a combination of the following values.

Value	Description
LMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **LocalDiscard** function to discard a block.
LMEM_DISCARDED	The object's memory block has been discarded.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalFlags, LocalAlloc, LocalDiscard, LocalLock, LocalReAlloc, LocalUnlock

---

1.268 LocalFree

The **LocalFree** function frees the specified local memory object and invalidates its handle.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalFree: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalFree@4" );
```

Parameters

hMem

[in] Handle to the local memory object. This handle is returned by either the **LocalAlloc** or **LocalReAlloc** function.

Return Values

If the function succeeds, the return value is NULL.

If the function fails, the return value is equal to a handle to the local memory object. To get extended error information, call **GetLastError**.

Remarks

If the process tries to examine or modify the memory after it has been freed, heap corruption may occur or an access violation exception (EXCEPTION_ACCESS_VIOLATION) may be generated.

If the *hMem* parameter is NULL, **LocalFree** ignores the parameter and returns NULL.

The **LocalFree** function will free a locked memory object. A locked memory object has a lock count greater than zero. The **LocalLock** function locks a local memory object and increments the lock count by one. The **LocalUnlock** function unlocks it and decrements the lock count by one. To get the lock count of a local memory object, use the **LocalFlags** function.

If an application is running under a debug version of the system, **LocalFree** will issue a message that tells you that a locked object is being freed. If you are debugging the application, **LocalFree** will enter a breakpoint just before freeing a locked object. This allows you to verify the intended behavior, then continue execution.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GlobalFree, LocalAlloc, LocalFlags,

1.269 LocalHandle

The **LocalHandle** function retrieves the handle associated with the specified pointer to a local memory object.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalHandle: procedure
(
    var pMem:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalHandle@4" );
```

Parameters

pMem

[in] Pointer to the first byte of the local memory object. This pointer is returned by the **LocalLock** function.

Return Values

If the function succeeds, the return value is a handle to the specified local memory object.

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

Remarks

When the **LocalAlloc** function allocates a local memory object with LMEM_MOVEABLE, it returns a handle to the object. The **LocalLock** function converts this handle into a pointer to the object's memory block, and **LocalHandle** converts the pointer back into a handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalAlloc, LocalLock

1.270 LocalLock

The **LocalLock** function locks a local memory object and returns a pointer to the first byte of the object's memory block.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalLock: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
```

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

Parameters

hMem

[in] Handle to the local memory object. This handle is returned by either the **LocalAlloc** or **LocalReAlloc** function.

Return Values

If the function succeeds, the return value is a pointer to the first byte of the memory block.

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

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, **LocalLock** increments the count by one, and the **LocalUnlock** function decrements the count by one. For each call that a process makes to **LocalLock** for an object, it must eventually call **LocalUnlock**. Locked memory will not be moved or discarded unless the memory object is reallocated by using the **LocalReAlloc** function. The memory block of a locked memory object remains locked in memory until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with LMEM_FIXED always have a lock count of zero. For these objects, the value of the returned pointer is equal to the value of the specified handle.

If the specified memory block has been discarded or if the memory block has a zero-byte size, this function returns NULL.

Discarded objects always have a lock count of zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalAlloc, LocalFlags, LocalReAlloc, LocalUnlock

---

1.271 LocalReAlloc

The **LocalReAlloc** function changes the size or the attributes of a specified local memory object. The size can increase or decrease.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalReAlloc: procedure
(
    hMem:       dword;
    uBytes:     SIZE_T;
    uFlags:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalReAlloc@12" );
```

Parameters

hMem

[in] Handle to the local memory object to be reallocated. This handle is returned by either the **LocalAlloc** or **LocalReAlloc** function.

uBytes

[in] New size, in bytes, of the memory block. If *uFlags* specifies LMEM_MODIFY, this parameter is ignored.

uFlags

[in] Specifies how to reallocate the local memory object. If LMEM_MODIFY is specified, this parameter modifies the attributes of the memory object, and the *uBytes* parameter is ignored. Otherwise, this parameter controls the reallocation of the memory object.

You can combine LMEM_MODIFY with the following value.

Value	Meaning
LMEM_DISCARDABLE	Ignored. This value is provided only for compatibility with 16-bit Windows. In Win32, you must explicitly call the **LocalDiscard** function to discard a block.

If this parameter does not specify LMEM_MODIFY, this parameter can be any combination of the following values.

Value	Meaning
LMEM_MOVEABLE	Allocates movable memory. Otherwise, the memory will only be reallocated in place. The return value is a handle to the memory object. To convert the handle to a pointer, use the **LocalLock** function.
LMEM_NOCOMPACT	Ignored. This value is provided only for compatibility with 16-bit Windows.
LMEM_ZEROINIT	Causes the additional memory contents to be initialized to zero if the memory object is growing in size.

Return Values

If the function succeeds, the return value is a handle to the reallocated memory object.

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

Remarks

If **LocalReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

If **LocalReAlloc** reallocates a fixed object, the value of the handle returned is the address of the first byte of the memory block. To access the memory, a process can simply cast the return value to a pointer.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalAlloc, LocalFree, LocalLock

---

1.272 LocalSize

The **LocalSize** function returns the current size, in bytes, of the specified local memory object.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalSize: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalSize@4" );
```

Parameters

hMem

[in] Handle to the local memory object. This handle is returned by the **LocalAlloc, LocalReAlloc**, or **LocalHandle** function.

Return Values

If the function succeeds, the return value is the size, in bytes, of the specified local memory object. If the specified handle is not valid or if the object has been discarded, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The size of a memory block may be larger than the size requested when the memory was allocated.

To verify that the specified object's memory block has not been discarded, call the **LocalFlags** function before calling **LocalSize**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalAlloc, LocalFlags, LocalHandle, LocalReAlloc

---

1.273 LocalUnlock

The **LocalUnlock** function decrements the lock count associated with a memory object that was allocated with LMEM_MOVEABLE. This function has no effect on memory objects allocated with LMEM_FIXED.

Note The local functions are slower than other memory management functions and do not provide as many features. Therefore, new applications should use the heap functions.

```
LocalUnlock: procedure
(
    hMem:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LocalUnlock@4" );
```

Parameters

hMem

[in] Handle to the local memory object. This handle is returned by either the **LocalAlloc** or **LocalReAlloc** function.

Return Values

If the memory object is still locked after decrementing the lock count, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. If **GetLastError** returns NO_ERROR, the memory object is unlocked.

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, the **LocalLock** function increments the count by one, and **LocalUnlock** decrements the count by one. For each call that a process makes to **LocalLock** for an object, it must eventually call **LocalUnlock**. Locked memory will not be moved or discarded unless the memory object is reallocated by using the **LocalReAlloc** function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

If the memory object is already unlocked, **LocalUnlock** returns FALSE and **GetLastError** reports ERROR_NOT_LOCKED. Memory objects allocated with LMEM_FIXED always have a lock count of zero and cause the ERROR_NOT_LOCKED error.

A process should not rely on the return value to determine the number of times it must subsequently call **LocalUnlock** for the memory block.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, LocalAlloc, LocalFlags, LocalLock, LocalReAlloc

---

1.274 LockFile

The **LockFile** function locks a region in an open file. Locking a region prevents other processes from accessing the region.

To specify additional options, use the **LockFileEx** function.

```
LockFile: procedure
(
```

```
        hFile:                      dword;
        dwFileOffsetLow:            dword;
        dwFileOffsetHigh:           dword;
        nNumberOfBytesToLockLow:    dword;
        nNumberOfBytesToLockHigh:   dword
    );
        stdcall;
        returns( "eax" );
        external( "__imp__LockFile@20" );
```

Parameters

hFile

[in] Handle to the file with a region to be locked. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file (or both).

dwFileOffsetLow

[in] Specifies the low-order word of the starting byte offset in the file where the lock should begin.

dwFileOffsetHigh

[in] Specifies the high-order word of the starting byte offset in the file where the lock should begin.

Windows 95/98: *dwFileOffsetHigh* must be 0, the sign extension of the value of *dwFileOffsetLow*. Any other value will be rejected.

nNumberOfBytesToLockLow

[in] Specifies the low-order word of the length of the byte range to be locked.

nNumberOfBytesToLockHigh

[in] Specifies the high-order word of the length of the byte range to be locked.

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

Locking a region of a file gives the locking process exclusive access to the specified region. File locks are not inherited by processes created by the locking process.

Locking a region of a file denies all other processes both read and write access to the specified region. Locking a region that goes beyond the current end-of-file position is not an error.

Locks may not overlap an existing locked region of the file.

If **LockFile** cannot lock a region of a file, it returns zero immediately. It does not block. To issue a file lock request that will block until the lock is acquired, use **LockFileEx** without LOCKFILE_FAIL_IMMEDIATELY.

The **UnlockFile** function unlocks a file region locked by **LockFile**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, LockFileEx, UnlockFile

1.275 LockFileEx

The **LockFileEx** function locks a region in an open file for shared or exclusive access. Locking a region prevents other processes from accessing the region.

```
LockFileEx: procedure
(
        hFile:                      dword;
        dwFlags:                    dword;
        dwReserved:                 dword;
        nNumberOfBytesToLockLow:    dword;
        nNumberOfBytesToLockHigh:   dword;
    var lpOverlapped:               OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__LockFileEx@24" );
```

Parameters

hFile

[in] Handle to an open handle to a file that is to have a range of bytes locked for shared or exclusive access. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwFlags

[in] Specifies flags that modify the behavior of this function. This parameter may be one or more of the following values.

Value	Meaning
LOCKFILE_FAIL_IMMEDIATELY	If this value is specified, the function returns immediately if it is unable to acquire the requested lock. Otherwise, it waits.
LOCKFILE_EXCLUSIVE_LOCK	If this value is specified, the function requests an exclusive lock. Otherwise, it requests a shared lock.

dwReserved

Reserved parameter; must be set to zero.

nNumberOfBytesToLockLow

[in] Specifies the low-order 32 bits of the length of the byte range to lock.

nNumberOfBytesToLockHigh

[in] Specifies the high-order 32 bits of the length of the byte range to lock.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure that the function uses with the locking request. This structure, which is required, contains the file offset of the beginning of the lock range. You must initialize the **hEvent** member to a valid handle or zero.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Locking a region of a file is used to acquire shared or exclusive access to the specified region of the file. File locks are not inherited by a new process during process creation.

Locking a portion of a file for exclusive access denies all other processes both read and write access to the specified region of the file. Locking a region that goes beyond the current end-of-file position is not an error.

Locking a portion of a file for shared access denies all processes write access to the specified region of the file, including the process that first locks the region. All processes can read the locked region.

The **LockFileEx** function operates asynchronously if the file handle was opened for asynchronous I/O, unless the LOCKFILE_FAIL_IMMEDIATELY flag is specified. If an exclusive lock is requested for a range of a file that already has a shared or exclusive lock, the function returns the error ERROR_IO_PENDING. The system will signal the event specified in the **OVERLAPPED** structure after the lock is granted. To determine when the lock has been granted, use the **GetOverlappedResult** function or one of the wait functions.

If the file handle was not opened for asynchronous I/O and the lock is not available, this call waits until the lock is granted or an error occurs, unless the LOCKFILE_FAIL_IMMEDIATELY flag is specified.

Locks may not overlap an existing locked region of the file.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, LockFile, OVERLAPPED, UnlockFile, UnlockFileEx

---

1.276 LockResource

The **LockResource** function locks the specified resource in memory.

```
LockResource: procedure
(
    hResData:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__LockResource@4" );
```

Parameters

hResData

[in] Handle to the resource to be locked. The **LoadResource** function returns this handle. Note that this parameter is listed as an HGLOBAL variable only for backwards compatibility. Do not pass any value as a parameter other than a successful return value from the **LoadResource** function.

Return Values

If the loaded resource is locked, the return value is a pointer to the first byte of the resource; otherwise, it is NULL.

Remarks

The pointer returned by **LockResource** is valid until the module containing the resource is unloaded. It is not necessary to unlock resources because the system automatically deletes them when the process that created them terminates.

Do not try to lock a resource by using the handle returned by the **FindResource** or **FindResourceEx** function. Such a handle points to random data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, FindResource, FindResourceEx, LoadResource

---

1.277 lstrcat

The **lstrcat** function appends one string to another.

```
lstrcat: procedure
(
    var lpString1:  var;
    var lpString2:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrcat@8" );
```

Parameters

lpString1

[in/out] Pointer to a null-terminated string. The buffer must be large enough to contain both strings.

lpString2

[in] Pointer 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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, lstrcmp, lstrcmpi, lstrcpy, lstrlen

---

1.278 lstrcmp

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

To perform a comparison that is not case sensitive, use the **lstrcmpi** function.

```
lstrcmp: procedure
(
    var lpString1:  var;
    var lpString2:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrcmp@8" );
```

Parameters

lpString1

[in] Pointer to the first null-terminated string to be compared.

lpString2

[in] Pointer to the second null-terminated string to be compared.

Return Values

If 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*, the return value 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 through Control Panel, determines which string is greater (or whether the strings are the same). If no language (locale) is selected, the system performs the comparison by using default values.

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

The **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 .

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, CompareString, lstrcat, lstrcmpi, lstrcpy, lstrlen

---

1.279 lstrcmpi

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

To perform a comparison that is case sensitive, use the **`lstrcmp`** function.

```
lstrcmpi: procedure
```

```
(
    var lpString1:  var;
    var lpString2:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrcmpi@8" );
```

Parameters

lpString1

[in] Pointer to the first null-terminated string to be compared.

lpString2

[in] Pointer to the second null-terminated string to be compared.

Return Values

If 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*, the return value 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 through Control Panel, determines which string is greater (or whether the strings are the same). If no language (locale) is selected, the system 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 the system, this function can compare two DBCS strings.

The **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 .

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, CompareString, lstrcat, lstrcmp, lstrcpy, lstrlen

1.280 lstrcpy

The **lstrcpy** function copies a string to a buffer.

To copy a specified number of characters, use the **lstrcpyn** function.

```
lstrcpy: procedure
(
    var lpString1:  var;
    var lpString2:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrcpy@8" );
```

Parameters

lpString1

[out] Pointer 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

[in] Pointer 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.

Remarks

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

The **lstrcpy** function has an undefined behavior if source and destination buffers overlap.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, lstrcat, lstrcmp, lstrcmpi, lstrlen

1.281 lstrcpyn

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

```
lstrcpyn: procedure
(
    var lpString1:  var;
    var lpString2:  var
        iMaxLength: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrcpyn@12" );
```

Parameters

lpString1

[out] Pointer to a buffer into which the function copies characters. The buffer must be large enough to contain the number of **TCHARs** specified by *iMaxLength*, including room for a terminating null character.

lpString2

[in] Pointer to a null-terminated string from which the function copies characters.

iMaxLength

[in] Specifies the number of **TCHARs** to be copied from the string pointed to by *lpString2* into the buffer pointed to by *lpString1*, including a terminating null character. This refers to bytes for ANSI versions of the function or characters for Unicode versions.

Return Values

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

If the function fails, the return value is NULL.

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.

The lstrcpyn function has an undefined behavior if source and destination buffers overlap.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, lstrcat, lstrcmp, lstrcmpi, lstrcpy, lstrlen

---

1.282 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).

```
lstrlen: procedure
(
    var lpString:   var
);
    stdcall;
    returns( "eax" );
    external( "__imp__lstrlen@4" );
```

Parameters

lpString

[in] Pointer to a null-terminated string.

Return Values

The return value specifies the length of the string, in **TCHARs**. This refers to bytes for ANSI versions of the function or characters for Unicode versions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Strings Overview, String Functions, lstrcat, lstrcmp, lstrcmpi, lstrcpy

---

1.283 MapViewOfFile

The **MapViewOfFile** function maps a view of a file into the address space of the calling process.

To specify a suggested base address, use the **MapViewOfFileEx** function.

```
MapViewOfFile: procedure
(
    hFileMappingObject:     dword;
    dwDesiredAccess:        dword;
    dwFileOffsetHigh:       dword;
    dwFileOffsetLow:        dword;
    dwNumberOfBytesToMap:   SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__MapViewOfFile@20" );
```

Parameters

hFileMappingObject

[in] Handle to an open handle of a file-mapping object. The **CreateFileMapping** and **OpenFileMapping** functions return this handle.

dwDesiredAccess

[in] Specifies the type of access to the file view and, therefore, the protection of the pages mapped by the file. This parameter can be one of the following values.

Value	Meaning
FILE_MAP_WRITE	Read/write access. The *hFileMappingObject* parameter must have been created with PAGE_READWRITE protection. A read/write view of the file is mapped.

FILE_MAP_READ	Read-only access. The *hFileMappingObject* parameter must have been created with PAGE_READWRITE or PAGE_READONLY protection. A read-only view of the file is mapped.
FILE_MAP_ALL_ACCESS	Same as FILE_MAP_WRITE.
FILE_MAP_COPY	Copy on write access. If you create the map with PAGE_WRITECOPY and the view with FILE_MAP_COPY, you will receive a view to file. If you write to it, the pages are automatically swappable and the modifications you make will not go to the original data file. **Windows 95:** You must pass PAGE_WRITECOPY to **CreateFileMapping**; otherwise, an error will be returned. If you share the mapping between multiple processes using **DuplicateHandle** or **OpenFileMapping** and one process writes to a view, the modification is propagated to the other process. The original file does not change. **Windows NT/2000:** There is no restriction as to how the *hFileMappingObject* parameter must be created. Copy on write is valid for any type of view. If you share the mapping between multiple processes using **DuplicateHandle** or **OpenFileMapping** and one process writes to a view, the modification is not propagated to the other process. The original file does not change.

dwFileOffsetHigh

[in] Specifies the high-order **DWORD** of the file offset where mapping is to begin.

dwFileOffsetLow

[in] Specifies the low-order **DWORD** of the file offset where mapping is to begin. The combination of the high and low offsets must specify an offset within the file that matches the system's memory allocation granularity, or the function fails. That is, the offset must be a multiple of the allocation granularity. Use the **GetSystemInfo** function, which fills in the members of a **SYSTEM_INFO** structure, to obtain the system's memory allocation granularity.

dwNumberOfBytesToMap

[in] Specifies the number of bytes of the file to map. If *dwNumberOfBytesToMap* is zero, the entire file is mapped.

Return Values

If the function succeeds, the return value is the starting address of the mapped view.

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

Remarks

Mapping a file makes the specified portion of the file visible in the address space of the calling process.

Multiple views of a file (or a file-mapping object and its mapped file) are said to be "coherent" if they contain identical data at a specified time. This occurs if the file views are derived from the same file-mapping object. A process can duplicate a file-mapping object handle into another process by using the **DuplicateHandle** function, or another process can open a file-mapping object by name by using the **OpenFileMapping** function.

A mapped view of a file is not guaranteed to be coherent with a file being accessed by the **ReadFile** or **WriteFile** function:

Windows 95: MapViewOfFile may require the swapfile to grow. If the swapfile cannot grow, the function fails.

Windows NT/2000: If the file-mapping object is backed by the paging file (*hFile* is INVALID_HANDLE_VALUE), the paging file must be large enough to hold the entire mapping. If it is not, **MapViewOfFile** fails.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or

reads from a memory mapped view. For more information, see Reading and Writing.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, CreateFileMapping, DuplicateHandle, GetSystemInfo, MapViewOfFileEx, OpenFileMapping, UnmapViewOfFile, SYSTEM_INFO

1.284 MapViewOfFileEx

The **MapViewOfFileEx** function maps a view of a file into the address space of the calling process. This extended function allows the calling process to specify a suggested memory address for the mapped view.

```
MapViewOfFileEx: procedure
(
        hFileMappingObject:     dword;
        dwDesiredAccess:        dword;
        dwFileOffsetHigh:       dword;
        dwFileOffsetLow:        dword;
        dwNumberOfBytesToMap:   SIZE_T;
    var lpBaseAddress:          var
);
    stdcall;
    returns( "eax" );
    external( "__imp__MapViewOfFileEx@24" );
```

Parameters

hFileMappingObject

[in] Handle to an open handle to a file-mapping object. The **CreateFileMapping** and **OpenFileMapping** functions return this handle.

dwDesiredAccess

[in] Specifies the type of access to the file-mapping object and, therefore, the page protection of the pages mapped by the file. This parameter can be one of the following values.

Value	Meaning
FILE_MAP_WRITE	Read-and-write access. The *hFileMappingObject* parameter must have been created with PAGE_READWRITE protection. A read/write view of the file is mapped.
FILE_MAP_READ	Read-only access. The *hFileMappingObject* parameter must have been created with PAGE_READWRITE or PAGE_READONLY protection. A read-only view of the file is mapped.
FILE_MAP_ALL_ACCESS	Same as FILE_MAP_WRITE.

FILE_MAP_COPY	Copy on write access. If you create the map with PAGE_WRITECOPY and the view with FILE_MAP_COPY, you will receive a view to the file. If you write to it, the pages are automatically swappable and the modifications you make will not go to the original data file. **Windows 95:** You must pass PAGE_WRITECOPY to **`CreateFileMapping`**; otherwise, an error will be returned. If you share the mapping between multiple processes using **`DuplicateHandle`** or **`OpenFileMapping`** and one process writes to a view, the modification is propagated to the other process. The original file does not change. **Windows NT/2000:** There is no restriction as to how the *hFileMappingObject* parameter must be created. Copy on write is valid for any type of view. If you share the mapping between multiple processes using **`DuplicateHandle`** or **`OpenFileMapping`** and one process writes to a view, the modification is not propagated to the other process. The original file does not change.

dwFileOffsetHigh
[in] Specifies the high-order **DWORD** of the file offset where mapping is to begin.

dwFileOffsetLow

[in] Specifies the low-order **DWORD** of the file offset where mapping is to begin. The combination of the high and low offsets must specify an offset within the file that matches the system's memory allocation granularity, or the function fails. That is, the offset must be a multiple of the allocation granularity. Use the **`GetSystemInfo`** function, which fills in the members of a **`SYSTEM_INFO`** structure, to obtain the system's memory allocation granularity.

dwNumberOfBytesToMap

[in] Specifies the number of bytes of the file to map. If *dwNumberOfBytesToMap* is zero, the entire file is mapped.

lpBaseAddress

[in] Pointer to the memory address in the calling process's address space where mapping should begin. This must be a multiple of the system's memory allocation granularity, or the function fails. Use the **`GetSystemInfo`** function, which fills in the members of a **`SYSTEM_INFO`** structure, to obtain the system's memory allocation granularity. If there is not enough address space at the specified address, the function fails.

If *lpBaseAddress* is NULL, the operating system chooses the mapping address. In this case, this function is equivalent to the **`MapViewOfFile`** function.

Return Values

If the function succeeds, the return value is the starting address of the mapped view.

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

Remarks

Mapping a file makes the specified portion of the file visible in the address space of the calling process.

If a suggested mapping address is supplied, the file is mapped at the specified address (rounded down to the nearest 64K boundary) if there is enough address space at the specified address. If there is not, the function fails.

Typically, the suggested address is used to specify that a file should be mapped at the same address in multiple processes. This requires the region of address space to be available in all involved processes. No other memory allocation, including use of the **`VirtualAlloc`** function to reserve memory, can take place in the region used for mapping:

Windows 95: If the *lpBaseAddress* parameter specifies a base offset, the function succeeds only if the same memory region is available for the memory mapped file in all other 32-bit processes.

Windows NT/2000: If the *lpBaseAddress* parameter specifies a base offset, the function succeeds if the given memory region is not already in use by the calling process. the system does *not* guarantee that the same

memory region is available for the memory mapped file in other 32-bit processes.

Multiple views of a file (or a file-mapping object and its mapped file) are said to be "coherent" if they contain identical data at a specified time. This occurs if the file views are derived from the same file-mapping object. A process can duplicate a file-mapping object handle into another process by using the **DuplicateHandle** function, or another process can open a file-mapping object by name by using the **OpenFileMapping** function.

A mapped view of a file is not guaranteed to be coherent with a file being accessed by the **ReadFile** or **WriteFile** function.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a memory mapped view. For more information, see Reading and Writing.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, CreateFileMapping, DuplicateHandle, GetSystemInfo, OpenFileMapping, ReadFile, UnmapViewOfFile, SYSTEM_INFO, VirtualAlloc, WriteFile

---

1.285 Module32First

Retrieves information about the first module associated with a process.

```
Module32First: procedure
(
        hSnapshot:  dword;
    var lpme:       MODULEENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Module32First@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpme

[in/out] Pointer to a **MODULEENTRY32** structure.

Return Values

Returns TRUE if the first entry of the module list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no modules exist or the snapshot does not contain module information.

Remarks

The calling application must set the **dwSize** member of **MODULEENTRY32** to the size, in bytes, of the structure. **Module32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other modules associated with the specified process, use the **Module32Next** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions, , CreateToolhelp32Snapshot, MODULEENTRY32, Module32Next

---

1.286 Module32Next

Retrieves information about the next module associated with a process or thread.

```
Module32Next: procedure
(
        hSnapshot:  dword;
    var lpme:       MODULEENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Module32Next@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpme

[out] Pointer to a **MODULEENTRY32** structure.

Return Values

Returns TRUE if the next entry of the module list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no more modules exist.

Remarks

To retrieve information about first module associated with a process, use the **Module32First** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hff
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.287 MoveFile

The **MoveFile** function moves an existing file or a directory, including its children.

To specify how to move the file, use the **MoveFileEx** function.

```
MoveFile: procedure
(
    lpExistingFileName: string;
    lpNewFileName:      string
);
    stdcall;
    returns( "eax" );
    external( "__imp__MoveFileA@8" );
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that names an existing file or directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpNewFileName

[in] Pointer to a null-terminated string that specifies the new name of a file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

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 **MoveFile** function will move (rename) either a file or a directory (including its children) either in the same directory or across directories. The one caveat is that the **MoveFile** function will fail on directory moves when the destination is on a different volume.

Windows 2000: The **MoveFile** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CopyFile, MoveFileEx, MoveFileWithProgress

1.288 MoveFileEx

The **MoveFileEx** function moves an existing file or directory.

The **MoveFileWithProgress** function is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

```
MoveFileEx: procedure
(
    lpExistingFileName: string;
    lpNewFileName:      string;
    dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__MoveFileExA@12" );
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that names an existing file or directory on the local machine. In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

If *dwflags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, the file cannot have the read-only attribute.

lpNewFileName

[in] Pointer to a null-terminated string that specifies the new name of *lpExistingFileName* on the local machine.

When moving a file, the destination can be on a different file system or drive. If the destination is on another drive, you must set the MOVEFILE_COPY_ALLOWED flag in *dwFlags*.

When moving a directory, the destination must be on the same drive.

If *dwFlags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, *lpNewFileName* can be NULL. In this case, **MoveFileEx** registers the *lpExistingFileName* file to be deleted when the system restarts. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

dwFlags

[in] Specifies how to move the file. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED	If the file is to be moved to a different volume, the function simulates the move by using the **CopyFile** and **DeleteFile** functions. This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT.
MOVEFILE_CREATE_HARDLINK	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups. This value can be used only if the process is in the context of a user who belongs to the administrator group or the LocalSystem account. This value cannot be used with MOVEFILE_COPY_ALLOWED.

MOVEFILE_FAIL_IF_NOT_TRACKABLE	**Windows 2000:** The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING	If a file named *lpNewFileName* exists, the function replaces its contents with the contents of the *lpExistingFileName* file. This value cannot be used if *lpNewFileName* or *lpExistingFileName* names a directory.
MOVEFILE_WRITE_THROUGH	The function does not return until the file has actually been moved on the disk. Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation. This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the *dwFlags* parameter specifies MOVEFILE_DELAY_UNTIL_REBOOT, **MoveFileEx** stores the locations of the files to be renamed at restart in the following registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

The function fails if it cannot access the registry.

The **PendingFileRenameOperations** value is of type **REG_MULTI_SZ**. Each rename operation stores a pair of NULL-terminated strings. The system uses these registry entries to complete the operations at restart in the same order that they were issued. For example, the following code fragment creates registry entries that delete *szDstFile* and rename *szSrcFile* to be *szDstFile* at restart:

```
MoveFileEx(szDstFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);

MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);
```

The system stores the following entries in **PendingFileRenameOperations**:

```
szDstFile\0\0

szSrcFile\0szDstFile\0\0
```

Because the actual move and deletion operations specified with the MOVEFILE_DELAY_UNTIL_REBOOT flag take place after the calling application has ceased running, the return value cannot reflect success or failure in moving or deleting the file. Rather, it reflects success or failure in placing the appropriate entries into the registry.

The system deletes a directory tagged for deletion with the MOVEFILE_DELAY_UNTIL_REBOOT flag only if it is empty. To ensure deletion of directories, move or delete all files from the directory before attempting to delete it. Files may be in the directory at boot time, but they must be deleted or moved before the system can delete the directory.

The move and deletion operations are carried out at boot time in the same order they are specified in the calling application. To delete a directory that has files in it at boot time, first delete the files.

Windows 2000: The **MoveFileEx** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

Windows 95/98: The **MoveFileEx** function is not supported. To rename or delete a file at restart, use the following procedure.

To rename or delete a file on Windows 95/98

Check for the existence of the WININIT.INI file in the Windows directory.

If WININIT.INI exists, open it and add new entries to the existing [rename] section. If the file does not exist, create the file and create a [rename] section.

Add lines of the following format to the [rename] section:

```
DestinationFileName=SourceFileName
```

Both *DestinationFileName* and *SourceFileName* must be short file names. To delete a file, use NUL as the value for *DestinationFileName*.

The system processes WININIT.INI during system boot. After WININIT.INI has been processed, the system names it WININIT.BAK.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CopyFile, DeleteFile, GetWindowsDirectory, MoveFileWithProgress, WritePrivateProfileString

---

1.289 MoveFileWithProgress

The **MoveFileWithProgress** function moves a file or directory. **MoveFileWithProgress** is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

```
MoveFileWithProgress: procedure
(
        lpExistingFileName: string;
        lpNewFileName:      string;
        lpProgressRoutine:  procedure;
    var lpData:             var;
        dwFlags:            dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__MoveFileWithProgressA@20" );
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that names an existing file or directory on the local machine.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

lpNewFileName

[in] Pointer to a null-terminated string containing the new name of the file or directory.

When moving a file, *lpNewFileName* can be on a different file system or drive. If *lpNewFileName* is on another drive, you must set the MOVEFILE_COPY_ALLOWED flag in *dwFlags*.

When moving a directory, *lpExistingFileName* and *lpNewFileName* must be on the same drive.

If *dwFlags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, *lpNewFileName* can be NULL. In this case, **MoveFileEx** registers *lpExistingFileName* to be deleted when the system restarts. The function fails if it cannot access the registry to store the information about the delete operation. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

lpProgressRoutine

[in] Pointer to a **`CopyProgressRoutine`** callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be NULL.

lpData

[in] Specifies an argument that **MoveFileWithProgress** passes to the **CopyProgressRoutine** callback function. This parameter can be NULL.

dwFlags

[in] Specifies how to move the file. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED	If the file is to be moved to a different volume, the function simulates the move by using the **`CopyFile`** and **`DeleteFile`** functions. This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT.
MOVEFILE_CREATE_HARDLINK	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups. This value can only be used if the process is in the context of a user who belongs to the administrator group or the LocalSystem account. This value cannot be used with MOVEFILE_COPY_ALLOWED.
MOVEFILE_FAIL_IF_NOT_TRACKABLE	The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING	If a file named *lpNewFileName* exists, the function replaces its contents with the contents of the *lpExistingFileName* file. This value cannot be used if *lpNewFileName* or *lpExistingFileName* names a directory.

MOVEFILE_WRITE_THROUGH	The function does not return until the file has actually been moved on the disk. Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation. This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.

Return Value

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 **MoveFileWithProgress** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CopyFileEx, CopyProgressRoutine, MoveFileEx

---

1.290 MulDiv

The **MulDiv** function multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value. The return value is rounded up or down to the nearest integer.

```
MulDiv: procedure
(
    nNumber:        dword;
    nNumerator:     dword;
    nDenominator:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__MulDiv@12" );
```

Parameters

nNumber

[in] Specifies the multiplicand.

nNumerator

[in] Specifies the multiplier.

nDenominator

[in] Specifies the number by which the result of the multiplication (*nNumber* * *nNumerator*) is to be divided.

Return Values

If the function succeeds, the return value is the result of the multiplication and division. If either an overflow occurred or *nDenominator* was 0, the return value is –1.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Int32x32To64, UInt32x32To64

1.291 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.

```
MultiByteToWideChar: procedure
(
        CodePage:       dword;
        dwFlags:        dword;
        lpMultiByteStr: string;
        cbMultiByte:    dword;
    var lpWideCharStr:  var;
        cchWideChar:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__MultiByteToWideChar@24" );
```

Parameters

CodePage

[in] Specifies the code page to be used to perform the conversion. This parameter can be given the value of any code page that is installed or available in the system. You can also specify one of the values shown in the following table.

Value	Meaning
CP_ACP	ANSI code page
CP_MACCP	Macintosh code page
CP_OEMCP	OEM code page
CP_SYMBOL	**Windows 2000:** Symbol code page (42)
CP_THREAD_ACP	**Windows 2000:** The current thread's ANSI code page

CP_UTF7	**Windows NT 4.0 and Windows 2000:** Translate using UTF-7
CP_UTF8	**Windows NT 4.0 and Windows 2000:** Translate using UTF-8. When this is set, *dwFlags* must be zero.

dwFlags

[in] Indicates 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.

Value	Meaning
MB_PRECOMPOSED	Always use precomposed characters—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—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/nonspacing 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

[in] Points to the character string to be converted.

cbMultiByte

[in] Specifies the size in bytes of the string pointed to by the *lpMultiByteStr* parameter, or it can be -1 if the string is null terminated.

If this parameter is -1, the function processes the entire input string including the null terminator. The resulting wide character string therefore has a null terminator, and the returned length includes the null terminator.

If this parameter is a positive integer, the function processes exactly the specified number of bytes. If the given length does not include a null terminator then the resulting wide character string will not be null terminated, and the returned length does not include a null terminator.

lpWideCharStr

[out] Points to a buffer that receives the translated string.

cchWideChar

[in] 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 encounters an invalid character in the source string. An invalid character is either, a) a character that is not the default character in the source string but translates to the default character when MB_ERR_INVALID_CHARS is *not* set, or b) for DBCS strings, a character which has a lead byte but no valid trailing byte. 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.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Functions, WideCharToMultiByte

---

1.292 OpenEvent

The **OpenEvent** function opens an existing named event object.

```
OpenEvent: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandle:     boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenEventA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the requested access to the event object. For systems that support object security, the function fails if the security descriptor of the specified object does not permit the requested access for the calling process.

This parameter can be any combination of the following values.

Access	Description
EVENT_ALL_ACCESS	Specifies all possible access flags for the event object.
EVENT_MODIFY_STATE	Enables use of the event handle in the **SetEvent** and *ResetEvent* functions to modify the event's state.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the event handle in any of the wait functions to wait for the event's state to be signaled.

bInheritHandle

[in] Specifies whether the returned handle is inheritable. If TRUE, a process created by the **CreateProcess** function can inherit the handle; otherwise, the handle cannot be inherited.

lpName

[in] Pointer to a null-terminated string that names the event to be opened. Name comparisons are case sensitive.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

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

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

Remarks

The **OpenEvent** function enables multiple processes to open handles of the same event object. The function succeeds only if some process has already created the event by using the **CreateEvent** function. The calling process can use the returned handle in any function that requires a handle to an event object, subject to the limitations of the access specified in the *dwDesiredAccess* parameter.

The handle can be duplicated by using the **DuplicateHandle** function. Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateEvent, CreateProcess, DuplicateHandle, PulseEvent, ResetEvent, SetEvent, Object Names

---

1.293 OpenFile

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

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications

should use the **CreateFile** function.

```
OpenFile: procedure
(
        lpFileName:     string;
    var lpReOpenBuff:   OFSTRUCT;
        uStyle:         dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenFile@12" );
```

Parameters

lpFileName

[in] Pointer 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 file names. Also, **OpenFile** does not support opening named pipes.

lpReOpenBuff

[out] Pointer 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

[in] Specifies the action to take. This parameter can be one or more of the following values.

Value	Meaning
OF_CANCEL	Ignored. In the Win32 API, OF_PROMPT 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 the system 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/2000:** 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/2000:** 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/2000:** 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/2000:** 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 file name and extension only, this function searches for a matching file in the following directories, in the order shown:

The directory from which the application loaded.

The current directory.

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

Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/2000: The 16-bit Windows system directory. There is no Win32 function that retrieves the path of this directory, but it is searched. The name of this directory is SYSTEM.

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

The directories that are listed in the PATH environment variable.

The *lpFileName* parameter cannot contain wildcard characters.

The 32-bit **OpenFile** function does not support the OF_SEARCH flag supported by the 16-bit Windows **OpenFile** function. The OF_SEARCH flag directs the system to search for a matching file even when the file name 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.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, GetSystemDirectory, GetWindowsDirectory, _lclose, OFSTRUCT, SearchPath

---

1.294 OpenFileMapping

The **OpenFileMapping** function opens a named file-mapping object.

```
OpenFileMapping: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandle:     boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenFileMappingA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the access to the file-mapping object.

Windows NT/2000: This access is checked against any security descriptor on the target file-mapping object.

Windows 95/98: Security descriptors on file-mapping objects are not supported.

This parameter can be one of the following values.

Value	Meaning
FILE_MAP_WRITE	Read-write access. The target file-mapping object must have been created with PAGE_READWRITE or PAGE_WRITE protection. Allows a read-write view of the file to be mapped.
FILE_MAP_READ	Read-only access. The target file-mapping object must have been created with PAGE_READWRITE or PAGE_READ protection. Allows a read-only view of the file to be mapped.
FILE_MAP_ALL_ACCESS	All access. The target file-mapping object must have been created with PAGE_READWRITE protection. Allows a read-write view of the file to be mapped.
FILE_MAP_COPY	Copy-on-write access. The target file-mapping object must have been created with PAGE_WRITECOPY protection. Allows a copy-on-write view of the file to be mapped.

bInheritHandle

[in] Specifies whether the returned handle is to be inherited by a new process during process creation. A value of TRUE indicates that the new process inherits the handle.

lpName

[in] Pointer to a string that names the file-mapping object to be opened. If there is an open handle to a file-map-

ping object by this name and the security descriptor on the mapping object does not conflict with the *dwDesiredAccess* parameter, the open operation succeeds.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is an open handle to the specified file-mapping object.

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

Remarks

The handle that **OpenFileMapping** returns can be used with any function that requires a handle to a file-mapping object.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hfh
Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, CreateFileMapping

---

1.295 OpenJobObject

The **OpenJobObject** function opens an existing job object.

```
OpenJobObject: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandles:    boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenJobObjectA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the desired access mode to the job object. This parameter can be one or more of the following values.

Value	Meaning
MAXIMUM_ALLOWED	Specifies maximum access rights to the job object that are valid for the caller.

JOB_OBJECT_ASSIGN_PROCESS	Specifies the assign process access right to the object. Allows processes to be assigned to the job.
JOB_OBJECT_SET_ATTRIBUTES	Specifies the set attribute access right to the object. Allows job object attributes to be set.
JOB_OBJECT_QUERY	Specifies the query access right to the object. Allows job object attributes and accounting information to be queried.
JOB_OBJECT_TERMINATE	Specifies the terminate access right to the object. Allows termination of all processes in the job object.
JOB_OBJECT_SET_SECURITY_ATTRIBUTES	Specifies the security attributes access right to the object. Allows security limitations on all processes in the job object to be set.
JOB_OBJECT_ALL_ACCESS	Specifies the full access right to the job object.

bInheritHandles

[in] Specifies whether the returned handle is inherited when a new process is created. If this parameter is TRUE, the new process inherits the handle.

lpName

[in] Pointer to a null-terminated string specifying the name of the job to be opened. Name comparisons are case sensitive.

Return Values

If the function succeeds, the return value is a handle to the job. The handle provides the requested access to the job.

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

Remarks

To associate a process with a job, use the **AssignProcessToJobObject** function.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, AssignProcessToJobObject

---

1.296 OpenMutex

The **OpenMutex** function opens an existing named mutex object.

```
OpenMutex: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandle:     boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenMutexA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the requested access to the mutex object. For systems that support object security, the function fails if the security descriptor of the specified object does not permit the requested access for the calling process.

This parameter can be any combination of the following values.

Access	Description
MUTEX_ALL_ACCESS	Specifies all possible access flags for the mutex object.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the mutex handle in any of the wait functions to acquire ownership of the mutex, or in the **ReleaseMutex** function to release ownership.

bInheritHandle

[in] Specifies whether the returned handle is inheritable. If TRUE, a process created by the **CreateProcess** function can inherit the handle; otherwise, the handle cannot be inherited.

lpName

[in] Pointer to a null-terminated string that names the mutex to be opened. Name comparisons are case sensitive.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

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

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

Remarks

The **OpenMutex** function enables multiple processes to open handles of the same mutex object. The function succeeds only if some process has already created the mutex by using the **CreateMutex** function. The calling process can use the returned handle in any function that requires a handle to a mutex object, such as the wait functions, subject to the limitations of the access specified in the *dwDesiredAccess* parameter.

The handle can be duplicated by using the **DuplicateHandle** function. Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

Example

For an example that uses **OpenMutex**, see Using Named Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateMutex, CreateProcess, DuplicateHan-

1.297 OpenProcess

The **OpenProcess** function opens an existing process object.

```
OpenProcess: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandle:     boolean;
    dwProcessId:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenProcess@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the access to the process object. For operating systems that support security checking, this access is checked against any security descriptor for the target process. This parameter can be STANDARD_RIGHTS_REQUIRED or one or more of the following values.

Value	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the **CreateRemoteThread** function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the **DuplicateHandle** function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the **GetExitCodeProcess** and **GetPriorityClass** functions to read information from the process object.
PROCESS_SET_QUOTA	Enables using the process handle in the **AssignProcessToJobObject** and **SetProcessWorkingSetSize** functions to set memory limits.
PROCESS_SET_INFORMATION	Enables using the process handle in the **SetPriorityClass** function to set the priority class of the process.
PROCESS_TERMINATE	Enables using the process handle in the **TerminateProcess** function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the **VirtualProtectEx** and **WriteProcessMemory** functions to modify the virtual memory of the process.

PROCESS_VM_READ	Enables using the process handle in the **ReadProcessMemory** function to read from the virtual memory of the process.
PROCESS_VM_WRITE	Enables using the process handle in the **WriteProcessMemory** function to write to the virtual memory of the process.
SYNCHRONIZE	**Windows NT/2000:** Enables using the process handle in any of the wait functions to wait for the process to terminate.

bInheritHandle

bInheritHandle

[in] Specifies whether the returned handle can be inherited by a new process created by the current process. If TRUE, the handle is inheritable.

dwProcessId

[in] Specifies the identifier of the process to open.

Return Values

If the function succeeds, the return value is an open handle to the specified process.

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

Remarks

The handle returned by the **OpenProcess** function can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

When you are finished with the handle, be sure to close it using the **CloseHandle** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, AssignProcessToJobObject, CloseHandle, CreateProcess, CreateRemoteThread, DuplicateHandle, GetCurrentProcess, GetCurrentProcessId, GetExitCodeProcess, GetPriorityClass, ReadProcessMemory, SetPriorityClass, SetProcessWorkingSetSize, TerminateProcess, VirtualProtectEx, WriteProcessMemory

---

1.298 OpenSemaphore

The **OpenSemaphore** function opens an existing named semaphore object.

```
OpenSemaphore: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandles:    boolean;
    lpName:             string
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__OpenSemaphoreA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the requested access to the semaphore object. For systems that support object security, the function fails if the security descriptor of the specified object does not permit the requested access for the calling process.

This parameter can be any combination of the following values.

Access	Description
SEMAPHORE_ALL_ACCESS	Specifies all possible access flags for the semaphore object.
SEMAPHORE_MODIFY_STATE	Enables use of the semaphore handle in the **ReleaseSemaphore** function to modify the semaphore's count.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the semaphore handle in any of the wait functions to wait for the semaphore's state to be signaled.

bInheritHandle

[in] Specifies whether the returned handle is inheritable. If TRUE, a process created by the **CreateProcess** function can inherit the handle; otherwise, the handle cannot be inherited.

lpName

[in] Pointer to a null-terminated string that names the semaphore to be opened. Name comparisons are case sensitive.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

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

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

Remarks

The **OpenSemaphore** function enables multiple processes to open handles of the same semaphore object. The function succeeds only if some process has already created the semaphore by using the **CreateSemaphore** function. The calling process can use the returned handle in any function that requires a handle to a semaphore object, such as the wait functions, subject to the limitations of the access specified in the *dwDesiredAccess* parameter.

The handle can be duplicated by using the **DuplicateHandle** function. Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CloseHandle, CreateSemaphore, DuplicateHandle, ReleaseSemaphore, Object Names

---

1.299 OpenWaitableTimer

The **OpenWaitableTimer** function opens an existing named waitable timer object.

```
OpenWaitableTimer: procedure
(
    dwDesiredAccess:    dword;
    bInheritHandles:    boolean;
    lpTimerName:        string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OpenWaitableTimerA@12" );
```

Parameters

dwDesiredAccess

[in] Specifies the requested access to the timer object. For systems that support object security, the function fails if the security descriptor of the specified object does not permit the requested access for the calling process.

This parameter can be any combination of the following values.

Value	Meaning
TIMER_ALL_ACCESS	Specifies all possible access rights for the timer object.
TIMER_MODIFY_STATE	Enables use of the timer handle in the **SetWaitableTimer** and **CancelWaitableTimer** functions to modify the timer's state.
TIMER_QUERY_STATE	Reserved for future use.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the timer handle in any of the wait functions to wait for the timer's state to be signaled.

bInheritHandle

[in] Specifies whether the returned handle is inheritable. If TRUE, a process created by the **CreateProcess** function can inherit the handle; otherwise, the handle cannot be inherited.

lpTimerName

[in] Pointer to a null-terminated string specifying the name of the timer object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Name Spaces.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Windows NT 4.0 and earlier, Windows 95/98: The name can contain any character except the backslash character.

Return Values

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

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

Remarks

The **OpenWaitableTimer** function enables multiple processes to open handles to the same timer object. The function succeeds only if some process has already created the timer using the **CreateWaitableTimer** function. The calling process can use the returned handle in any function that requires the handle to a timer object, such as the wait functions, subject to the limitations of the access specified in the *dwDesiredAccess* parameter.

The returned handle can be duplicated by using the **DuplicateHandle** function. Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The timer object is destroyed when its last handle has been closed.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CancelWaitableTimer, CloseHandle, CreateProcess, CreateWaitableTimer, DuplicateHandle, SetWaitableTimer, Object Names

---

1.300 OutputDebugString

The **OutputDebugString** function sends a string to the debugger for display.

```
OutputDebugString: procedure
(
    lpOutputString: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__OutputDebugStringA@4" );
```

Parameters

lpOutputString

[in] Pointer to the null-terminated string to be displayed.

Return Values

This function does not return a value.

Remarks

If the application has no debugger, the system debugger displays the string. If the application has no debugger and the system debugger is not active, **OutputDebugString** does nothing.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions

1.301 PeekConsoleInput

The **PeekConsoleInput** function reads data from the specified console input buffer without removing it from the buffer.

```
PeekConsoleInput: procedure
(
        hConsoleInput:          dword;
    var lpBuffer:               INPUT_RECORD;
        nLength:                dword;
    var lpNumberOfEventsRead:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__PeekConsoleInputA@16" );
```

Parameters

hConsoleInput

[in] Handle to the input buffer. The handle must have GENERIC_READ access.

lpBuffer

[out] Pointer to an **INPUT_RECORD** buffer that receives the input buffer data.

nLength

[in] Specifies the size, in records, of the buffer pointed to by the *lpBuffer* parameter.

lpNumberOfEventsRead

[out] Pointer to a variable that receives the number of input records read.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of records requested exceeds the number of records available in the buffer, the number available is read. If no data is available, the function returns immediately.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, INPUT_RECORD, ReadConsoleInput, Set-

1.302 PostQueuedCompletionStatus

The **PostQueuedCompletionStatus** function posts an I/O completion packet to an I/O completion port. The I/O completion packet will satisfy an outstanding call to the **GetQueuedCompletionStatus** function. The **GetQueuedCompletionStatus** function returns with the three values passed as the second, third, and fourth parameters of the call to **PostQueuedCompletionStatus**.

```
PostQueuedCompletionStatus: procedure
(
        CompletionPort:             dword;
        dwNumberOfBytesTransferred: dword;
        dwCompletionKey:            dword;
    var lpOverlapped:               OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__PostQueuedCompletionStatus@16" );
```

Parameters

CompletionPort

[in] Handle to an I/O completion port to which the I/O completion packet is to be posted.

dwNumberOfBytesTransferred

[in] Specifies a value to be returned through the *lpNumberOfBytesTransferred* parameter of the **GetQueuedCompletionStatus** function.

dwCompletionKey

[in] Specifies a value to be returned through the *lpCompletionKey* parameter of the **GetQueuedCompletionStatus** function.

lpOverlapped

[in] Specifies a value to be returned through the *lpOverlapped* parameter of the **GetQueuedCompletionStatus** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

For more information concerning *dwNumberOfBytesTransferred*, *dwCompletionKey*, and *lpOverlapped*, see **GetQueuedCompletionStatus** and the descriptions of the parameters those values are returned through.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateIoCompletionPort, GetQueuedCompletionStatus, OVERLAPPED

1.303 PrepareTape

The **PrepareTape** function prepares the tape to be accessed or removed.

```
PrepareTape: procedure
(
    hDevice:        dword;
    dwOperation:    dword;
    bImmediate:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__PrepareTape@12" );
```

Parameters

hDevice

[in] Handle to the device preparing the tape. This handle is created by using the CreateFile function.

dwOperation

[in] Specifies how the tape device is to be prepared. This parameter can be one of the following values.

Value	Meaning
TAPE_FORMAT	Performs a low-level format of the tape. Currently, only the QIC117 device supports this feature.
TAPE_LOAD	Loads the tape and moves the tape to the beginning.
TAPE_LOCK	Locks the tape ejection mechanism so that the tape is not ejected accidentally.
TAPE_TENSION	Adjusts the tension by moving the tape to the end of the tape and back to the beginning. This option is not supported by all devices. This value is ignored if it is not supported.
TAPE_UNLOAD	Moves the tape to the beginning for removal from the device. After a successful unload operation, the device returns errors to applications that attempt to access the tape, until the tape is loaded again.
TAPE_UNLOCK	Unlocks the tape ejection mechanism.

bImmediate

[in] Specifies whether to return as soon as the preparation begins. If this parameter is TRUE, the function returns immediately. If it is FALSE, the function does not return until the operation has been completed.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.

ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

Some tape devices do not support certain tape operations. See your tape device documentation and use the **GetTapeParameters** function to determine your tape device's capabilities.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, GetTapeParameters

---

1.304 Process32First

Retrieves information about the first process encountered in a system snapshot.

```
Process32First: procedure
(
        hSnapshot:  dword;
    var lppe:       PROCESSENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Process32First@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lppe

[in/out] Pointer to a **PROCESSENTRY32** structure.

Return Values

Returns TRUE if the first entry of the process list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no processes exist or the snapshot does not contain process information.

Remarks

The calling application must set the **dwSize** member of **PROCESSENTRY32** to the size, in bytes, of the structure. **Process32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other processes recorded in the same snapshot, use the **Process32Next** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions, , CreateToolhelp32Snapshot, PROCESSENTRY32, Process32Next

---

1.305 Process32Next

Retrieves information about the next process recorded in a system snapshot.

```
Process32Next: procedure
(
        hSnapshot:  dword;
    var lppe:       PROCESSENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Process32Next@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lppe

[out] Pointer to a **PROCESSENTRY32** structure.

Return Values

Returns TRUE if the next entry of the process list has been copied to the buffer or FALSE otherwise. The

ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no processes exist or the snapshot does not contain process information.

Remarks

To retrieve information about the first process recorded in a snapshot, use the **Process32First** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.306 PulseEvent

The **PulseEvent** function sets the specified event object to the signaled state and then resets it to the nonsignaled state after releasing the appropriate number of waiting threads.

```
PulseEvent: procedure
(
    hEvent: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__PulseEvent@4" );
```

Parameters

hEvent

[in] Handle to the event object. The **CreateEvent** or **OpenEvent** function returns this handle.

Windows NT/2000: The handle must have EVENT_MODIFY_STATE access. For more information, see Synchronization Object Security and Access Rights.

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

For a manual-reset event object, all waiting threads that can be released immediately are released. The function then resets the event object's state to nonsignaled and returns.

For an auto-reset event object, the function resets the state to nonsignaled and returns after releasing a single waiting thread, even if multiple threads are waiting.

If no threads are waiting, or if no thread can be released immediately, **PulseEvent** simply sets the event object's state to nonsignaled and returns.

Note that for a thread using the multiple-object wait functions to wait for all specified objects to be signaled, **PulseEvent** can set the event object’s state to signaled and reset it to nonsignaled without causing the wait function to return. This happens if not all of the specified objects are simultaneously signaled.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateEvent, OpenEvent, ResetEvent, SetEvent

---

1.307 PurgeComm

The **PurgeComm** function discards all characters from the output or input buffer of a specified communications resource. It can also terminate pending read or write operations on the resource.

```
PurgeComm: procedure
(
    hFile:      dword;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__PurgeComm@8" );
```

Parameters

hFile

[in] Handle to the communications resource. The **CreateFile** function returns this handle.

dwFlags

[in] Specifies the action to take. This parameter can be one or more of the following values.

Value	Meaning
PURGE_TXABORT	Terminates all outstanding overlapped write operations and returns immediately, even if the write operations have not been completed.
PURGE_RXABORT	Terminates all outstanding overlapped read operations and returns immediately, even if the read operations have not been completed.
PURGE_TXCLEAR	Clears the output buffer (if the device driver has one).
PURGE_RXCLEAR	Clears the input buffer (if the device driver has one).

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If a thread uses **PurgeComm** to flush an output buffer, the deleted characters are not transmitted. To empty the output buffer while ensuring that the contents are transmitted, call the **FlushFileBuffers** function (a synchronous operation). Note, however, that **FlushFileBuffers** is subject to flow control but not to write time-outs, and it will not return until all pending write operations have been transmitted.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile

1.308 QueryDosDevice

The **QueryDosDevice** function retrieves information about MS-DOS device names. The function can obtain the current mapping for a particular MS-DOS device name. The function can also obtain a list of all existing MS-DOS device names.

MS-DOS device names are stored as symbolic links in the object name space. The code that converts an MS-DOS path into a corresponding path uses these symbolic links to map MS-DOS devices and drive letters. The **QueryDosDevice** function provides a mechanism whereby a Win32-based application can query the names of the symbolic links used to implement the MS-DOS device namespace as well as the value of each specific symbolic link.

```
QueryDosDevice: procedure
(
    lpDeviceName:   string;
    lpTargetPath:   string;
    ucchMax:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__QueryDosDeviceA@12" );
```

Parameters

lpDeviceName

[in] Pointer to an MS-DOS device name string specifying the target of the query. The device name cannot have a trailing backslash.

This parameter can be NULL. In that case, the **QueryDosDevice** function will store a list of all existing MS-DOS device names into the buffer pointed to by *lpTargetPath*.

lpTargetPath

[out] Pointer to a buffer that will receive the result of the query. The function fills this buffer with one or more null-terminated strings. The final null-terminated string is followed by an additional NULL.

If *lpDeviceName* is non-NULL, the function retrieves information about the particular MS-DOS device specified by *lpDeviceName*. The first null-terminated string stored into the buffer is the current mapping for the device. The other null-terminated strings represent undeleted prior mappings for the device.

If *lpDeviceName* is NULL, the function retrieves a list of all existing MS-DOS device names. Each null-terminated string stored into the buffer is the name of an existing MS-DOS device.

ucchMax

[in] Specifies the maximum number of **TCHARs** that can be stored into the buffer pointed to by *lpTargetPath*.

Return Values

If the function succeeds, the return value is the number of **TCHARs** stored into the buffer pointed to by *lpTargetPath*.

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

Remarks

The **DefineDosDevice** function provides a means whereby a Win32-based application can create and modify the sym-

bolic links used to implement the MS-DOS device namespace.

MS-DOS device names are global. After is it defined, an MS-DOS device name remains visible to all processes until either it is explicitly removed or the system restarts.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, DefineDosDevice

1.309 QueryInformationJobObject

The **QueryInformationJobObject** function retrieves limit and job state information from the job object.

```
QueryInformationJobObject: procedure
(
        hJob:                   dword;
        vJobObjectInfoClass:    JOBOBJECTINFOCLASS;
    var lpJobObjectInfo:        var;
        cbJobObjectInfoLength:  dword;
    var lpReturnLength:         dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__QueryInformationJobObject@20" );
```

Parameters

hJob

[in] Handle to the job whose information is being queried. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_QUERY access right associated with it. For more information, see Job Object Security and Access Rights.

If this value is NULL and the calling process is associated with a job, the job associated with the calling process is used.

JobObjectInfoClass

[in] Specifies the information class for limits to be queried. This parameter can be one of the following values.

Value	Meaning
JobObjectBasicAccountingInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION** structure.
JobObjectBasicAndIoAccountingInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION** structure.
JobObjectBasicLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JobObjectBasicProcessIdList	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_PROCESS_ID_LIST** structure.

JobObjectBasicUIRestrictions	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_UI_RESTRICTIONS** structure.
JobObjectExtendedLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure.
JobObjectSecurityLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_SECURITY_LIMIT_INFORMATION** structure.

lpJobObjectInfo

[out] Receives the limit information. The format of this data depends on the value of the *JobObjectInfoClass* parameter.

cbJobObjectInfoLength

[in] Specifies the count, in bytes, of the job information being queried.

lpReturnLength

[out] Pointer to a variable that receives the length of data written to the structure pointed to by the *lpJobObject-Info* parameter. If you do not want to receive this information, specify NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

You can use **QueryInformationJobObject** to obtain the current limits, modify them, then use the **SetInformation-JobObject** function to set new limits.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, JOBOBJECT_BASIC_ACCOUNTING_INFORMATION, JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION, JOBOBJECT_BASIC_LIMIT_INFORMATION, JOBOBJECT_BASIC_PROCESS_ID_LIST, JOBOBJECT_BASIC_UI_RESTRICTIONS, JOBOBJECT_EXTENDED_LIMIT_INFORMATION, JOBOBJECT_SECURITY_LIMIT_INFORMATION, SetInformationJobObject

---

1.310 QueryPerformanceCounter

The **QueryPerformanceCounter** function retrieves the current value of the high-resolution performance counter, if one exists.

```
QueryPerformanceCounter: procedure
(
    var lpPerformanceCount: qword
);
    stdcall;
    returns( "eax" );
    external( "__imp__QueryPerformanceCounter@4" );
```

Parameters

lpPerformanceCount

[out] Pointer to a variable that receives the current performance-counter value, in counts. If the installed hardware does not support a high-resolution performance counter, this parameter can be zero.

Return Values

If the installed hardware supports a high-resolution performance counter, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. For example, if the installed hardware does not support a high-resolution performance counter, the function fails.

Remarks

On a multiprocessor machine, it should not matter which processor is called. However, you can get different results on different processors due to bugs in the BIOS or the HAL. To specify processor affinity for a thread, use the **SetThreadAffinityMask** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Timers Overview, Timer Functions, QueryPerformanceFrequency

1.311 QueryPerformanceFrequency

The **QueryPerformanceFrequency** function retrieves the frequency of the high-resolution performance counter, if one exists. The frequency cannot change while the system is running.

```
QueryPerformanceFrequency: procedure
(
    var lpPerformanceCount: qword
);
    stdcall;
    returns( "eax" );
    external( "__imp__QueryPerformanceFrequency@4" );
```

Parameters

lpFrequency

[out] Pointer to a variable that receives the current performance-counter frequency, in counts per second. If the installed hardware does not support a high-resolution performance counter, this parameter can be zero.

Return Values

If the installed hardware supports a high-resolution performance counter, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. For example, if the installed hardware does not support a high-resolution performance counter, the function fails.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Timers Overview, Timer Functions, QueryPerformanceCounter

1.312 QueueUserAPC

The **QueueUserAPC** function adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread.

```
QueueUserAPC: procedure
(
    pfnAPC:     pointer;
    hThread:    dword;
    dwData:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__QueueUserAPC@12" );
```

Parameters

pfnAPC

[in] Pointer to the application-supplied APC function to be called when the specified thread performs an alertable wait operation. For more information, see **APCProc**.

hThread

[in] Specifies the handle to the thread. The handle must have THREAD_SET_CONTEXT access. For more information, see Synchronization Object Security and Access Rights.

dwData

[in] Specifies a single value that is passed to the APC function pointed to by the *pfnAPC* parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. There are no error values defined for this function that can be retrieved by calling **GetLastError**.

Remarks

The APC support provided in the operating system allows an application to queue an APC object to a thread. Each thread has its own APC queue. The queuing of an APC is a request for the thread to call the APC function. The operating system issues a software interrupt to direct the thread to call the APC function.

When a user-mode APC is queued, the thread is not directed to call the APC function unless it is in an alertable state. After the thread is in an alertable state, the thread handles all pending APCs in first in, first out (FIFO) order, and the wait operation returns WAIT_IO_COMPLETION. A thread enters an alertable state by using **SleepEx, SignalObjectAndWait, WaitForSingleObjectEx, WaitForMultipleObjectsEx,** or **MsgWaitForMultipleObjectsEx** to perform an alertable wait operation.

If an application queues an APC before the thread begins running, the thread begins by calling the APC function. After the thread calls an APC function, it calls the APC functions for all APCs in its APC queue.

When the thread is terminated using the **ExitThread** or **TerminateThread** function, the APCs in its APC queue are

lost. The APC functions are not called.

Note that the **ReadFileEx**, **SetWaitableTimer**, and **WriteFileEx** functions are implemented using an APC as the completion notification callback mechanism.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, APCProc, MsgWaitForMultipleObjectsEx, ReadFileEx, SetWaitableTimer, SleepEx, WaitForMultipleObjectsEx, WaitForSingleObjectEx, WriteFileEx

---

1.313 RaiseException

The **RaiseException** function raises an exception in the calling thread.

```
RaiseException: procedure
(
        dwExceptionCode:dword;
        dwExceptionFlags:dword;
        nNumberOfArguments:dword;
    var lpArguments:dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__RaiseException@16" );
```

Parameters

dwExceptionCode

[in] Specifies the application-defined exception code of the exception being raised. The filter expression and exception-handler block of an exception handler can use the **GetExceptionCode** function to retrieve this value.

Note that the system will clear bit 28 of *dwExceptionCode* before displaying a message This bit is a reserved exception bit, used by the system for its own purposes.

dwExceptionFlags

[in] Specifies the exception flags. This can be either zero to indicate a continuable exception, or EXCEPTION_NONCONTINUABLE to indicate a noncontinuable exception. Any attempt to continue execution after a noncontinuable exception causes the EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

nNumberOfArguments

[in] Specifies the number of arguments in the *lpArguments* array. This value must not exceed EXCEPTION_MAXIMUM_PARAMETERS. This parameter is ignored if *lpArguments* is NULL.

lpArguments

[in] Pointer to an array of arguments. This parameter can be NULL. These arguments can contain any application-defined data that needs to be passed to the filter expression of the exception handler.

Return Values

This function does not return a value.

Remarks

The **RaiseException** function enables a process to use structured exception handling to handle private, software-generated, application-defined exceptions.

Raising an exception causes the exception dispatcher to go through the following search for an exception handler:

The system first attempts to notify the process's debugger, if any.

If the process is not being debugged, or if the associated debugger does not handle the exception, the system attempts to locate a frame-based exception handler by searching the stack frames of the thread in which the exception occurred. The system searches the current stack frame first, then proceeds backward through preceding stack frames.

If no frame-based handler can be found, or no frame-based handler handles the exception, the system makes a second attempt to notify the process's debugger.

If the process is not being debugged, or if the associated debugger does not handle the exception, the system provides default handling based on the exception type. For most exceptions, the default action is to call the **ExitProcess** function.

The values specified in the *dwExceptionCode*, *dwExceptionFlags*, *nNumberOfArguments*, and *lpArguments* parameters can be retrieved in the filter expression of a frame-based exception handler by calling the **GetExceptionInformation** function. A debugger can retrieve these values by calling the **WaitForDebugEvent** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Structured Exception Handling Overview, Structured Exception Handling Functions, ExitProcess, GetExceptionCode, GetExceptionInformation, WaitForDebugEvent

---

1.314 ReadConsole

The **ReadConsole** function reads character input from the console input buffer and removes it from the buffer.

```
ReadConsole: procedure
(
        hConsoleInput:          dword;
    var lpBuffer:               var;
        nNumberOfCharsToRead:   dword;
    var lpNumberOfCharsRead:    dword;
    var lpReserved:             var
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadConsoleA@20" );
```

Parameters

hConsoleInput

[in] Handle to the console input buffer. The handle must have GENERIC_READ access.

lpBuffer

[out] Pointer to a buffer that receives the data read from the console input buffer.

nNumberOfCharsToRead

[in] Specifies the number of **TCHARs** to read. Because the function can read either Unicode or ANSI characters, the size of the buffer pointed to by the *lpBuffer* parameter should be at least *nNumberOfCharsToRead* * sizeof(TCHAR) bytes.

lpNumberOfCharsRead

[out] Pointer to a variable that receives the number of **TCHARs** actually read.

lpReserved

[in] Reserved; must be NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

ReadConsole reads keyboard input from a console's input buffer. It behaves like the **ReadFile** function, except that it can read in either Unicode (wide-character) or ANSI mode. To have applications that maintain a single set of sources compatible with both modes, use **ReadConsole** rather than **ReadFile**. Although **ReadConsole** can only be used with a console input buffer handle, **ReadFile** can be used with other handles (such as files or pipes). **ReadConsole** fails if used with a standard handle that has been redirected to be something other than a console handle.

All of the input modes that affect the behavior of **ReadFile** have the same effect on **ReadConsole**. To retrieve and set the input modes of a console input buffer, use the **GetConsoleMode** and **SetConsoleMode** functions.

If the input buffer contains input events other than keyboard events (such as mouse events or window-resizing events), they are discarded. Those events can only be read by using the **ReadConsoleInput** function.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleMode, ReadConsoleInput, ReadFile, SetConsoleCP, SetConsoleMode, SetConsoleOutputCP, WriteConsole

---

1.315 ReadConsoleInput

The **ReadConsoleInput** function reads data from a console input buffer and removes it from the buffer.

```
ReadConsoleInput: procedure
(
        hConsoleInput:          dword;
    var lpBuffer:               INPUT_RECORD;
        nLength:                dword;
    var lpNumberOfEventsRead:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadConsoleInputA@16" );
```

Parameters

hConsoleInput

[in] Handle to the input buffer. The handle must have GENERIC_READ access.

lpBuffer

[out] Pointer to an **INPUT_RECORD** buffer that receives the input buffer data.

nLength

[in] Specifies the size, in input records, of the buffer pointed to by the *lpBuffer* parameter.

lpNumberOfEventsRead

[out] Pointer to a variable that receives the number of input records read.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of records requested in the *nLength* parameter exceeds the number of records available in the buffer, the number available is read. The function does not return until at least one input record has been read.

A process can specify a console input buffer handle in one of the wait functions to determine when there is unread console input. When the input buffer is not empty, the state of a console input buffer handle is signaled.

To determine the number of unread input records in a console's input buffer, use the **GetNumberOfConsoleInputEvents** function. To read input records from a console input buffer without affecting the number of unread records, use the **PeekConsoleInput** function. To discard all unread records in a console's input buffer, use the **FlushConsoleInputBuffer** function.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, FlushConsoleInputBuffer, GetNumberOfConsoleInputEvents, INPUT_RECORD, PeekConsoleInput, ReadConsole, ReadFile, SetConsoleCP, SetConsoleOutputCP, WriteConsoleInput

---

1.316 ReadConsoleOutput

The **ReadConsoleOutput** function reads character and color attribute data from a rectangular block of character cells in a console screen buffer, and the function writes the data to a rectangular block at a specified location in the destination buffer.

```
ReadConsoleOutput: procedure
(
        hConsoleOutput: dword;
    var lpBuffer:       CHAR_INFO;
```

```
        dwBufferSize:   COORD;
        dwBufferCoord:  COORD;
    var lpReadRegion:   SMALL_RECT
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadConsoleOutputA@20" );
```

Parameters

hConsoleOutput

[in] Handle to the screen buffer. The handle must have GENERIC_READ access.

lpBuffer

[out] Pointer to a destination buffer that receives the data read from the screen buffer. This pointer is treated as the origin of a two-dimensional array of **CHAR_INFO** structures whose size is specified by the *dwBufferSize* parameter.

dwBufferSize

[in] Specifies the size, in character cells, of the *lpBuffer* parameter. The **X** member of the **COORD** structure is the number of columns; the **Y** member is the number of rows.

dwBufferCoord

[in] Specifies the coordinates of the upper-left cell in the *lpBuffer* parameter that receives the data read from the screen buffer. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpReadRegion

[in/out] Pointer to a **SMALL_RECT** structure. On input, the structure members specify the upper-left and lower-right coordinates of the screen buffer rectangle from which the function is to read. On output, the structure members specify the actual rectangle that the function copied from.

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

ReadConsoleOutput treats the screen buffer and the destination buffer as two-dimensional arrays (columns and rows of character cells). The rectangle pointed to by the *lpReadRegion* parameter specifies the size and location of the block to be read from the screen buffer. A destination rectangle of the same size is located with its upper-left cell at the coordinates of the *dwBufferCoord* parameter in the *lpBuffer* array. Data read from the cells in the screen buffer source rectangle is copied to the corresponding cells in the destination buffer. If the corresponding cell is outside the boundaries of the destination buffer rectangle (whose dimensions are specified by the *dwBufferSize* parameter), the data is not copied.

Cells in the destination buffer corresponding to coordinates that are not within the boundaries of the screen buffer are left unchanged. In other words, these are the cells for which no screen buffer data is available to be read.

Before **ReadConsoleOutput** returns, it sets the members of the structure pointed to by the *lpReadRegion* parameter to the actual screen buffer rectangle whose cells were copied into the destination buffer. This rectangle reflects the cells in the source rectangle for which there existed a corresponding cell in the destination buffer, because **ReadConsoleOutput** clips the dimensions of the source rectangle to fit the boundaries of the screen buffer.

If the rectangle specified by *lpReadRegion* lies completely outside the boundaries of the screen buffer, or if the corresponding rectangle is positioned completely outside the boundaries of the destination buffer, no data is copied. In this case, the function returns with the members of the structure pointed to by the *lpReadRegion* parameter set such that the **Right** member is less than the **Left**, or the **Bottom** member is less than the **Top**. To determine the size of the screen buffer, use the **GetConsoleScreenBufferInfo** function.

The **ReadConsoleOutput** function has no effect on the screen buffer's cursor position. The contents of the screen buffer are not changed by the function.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the SetConsoleCP or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CHAR_INFO, COORD, ReadConsoleOutputAttribute, ReadConsoleOutputCharacter, SetConsoleCP, SetConsoleOutputCP, SMALL_RECT, WriteConsoleOutput

---

1.317 ReadConsoleOutputAttribute

The **ReadConsoleOutputAttribute** function copies a specified number of foreground and background color attributes from consecutive cells of a console screen buffer, beginning at a specified location.

```
ReadConsoleOutputAttribute: procedure
(
        hConsoleOutput:         dword;
    var lpAttribute:            word;
        nLength:                dword;
        dwReadCoord:            COORD;
    var lpNumberOfAttrsRead:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadConsoleOutputAttribute@20" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_READ access.

lpAttribute

[out] Pointer to a buffer that receives the attributes read from the screen buffer.

nLength

[in] Specifies the number of screen buffer character cells from which to read. The size of the buffer pointed to by the *lpAttribute* parameter should be *nLength* * sizeof(WORD).

dwReadCoord

[in] Specifies the coordinates of the first cell in the screen buffer from which to read. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpNumberOfAttrsRead

[out] Pointer to a variable that receives the number of attributes actually read.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of attributes to be read from extends beyond the end of the specified screen buffer row, attributes are read from the next row. If the number of attributes to be read from extends beyond the end of the screen buffer, attributes up to the end of the screen buffer are read.

Each attribute specifies the foreground (text) and background colors in which that character cell is drawn. The attribute values are some combination of the following values: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces red text on a white background:

```
FOREGROUND_RED | BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, COORD, ReadConsoleOutput, ReadConsoleOutputCharacter, WriteConsoleOutput, WriteConsoleOutputAttribute, WriteConsoleOutputCharacter

---

1.318 ReadConsoleOutputCharacter

The **ReadConsoleOutputCharacter** function copies a number of characters from consecutive cells of a console screen buffer, beginning at a specified location.

```
ReadConsoleOutputCharacter: procedure
(
        hConsoleOutput:         dword;
        lpCharacter:            string;
        nLength:                dword;
        dwReadCoord:            COORD;
    var lpNumberOfCharsRead:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadConsoleOutputCharacterA@20" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_READ access.

lpCharacter

[out] Pointer to a buffer that receives the characters read from the screen buffer.

nLength

[in] Specifies the number of screen buffer character cells from which to read. The size of the buffer pointed to by

the *lpCharacter* parameter should be *nLength* * sizeof(TCHAR).

dwReadCoord

[in] Specifies the coordinates of the first cell in the screen buffer from which to read. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpNumberOfCharsRead

[out] Pointer to a variable that receives the number of characters actually read.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of characters to be read from extends beyond the end of the specified screen buffer row, characters are read from the next row. If the number of characters to be read from extends beyond the end of the screen buffer, characters up to the end of the screen buffer are read.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, COORD, ReadConsoleOutput, ReadConsoleOutputAttribute, SetConsoleCP, SetConsoleOutputCP, WriteConsoleOutput, WriteConsoleOutputAttribute, WriteConsoleOutputCharacter

---

1.319 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.

This function is designed for both synchronous and asynchronous operation. The **ReadFileEx** function is designed solely for asynchronous operation. It lets an application perform other processing during a file read operation.

```
ReadFile: procedure
(
        hFile:                  dword;
    var lpBuffer:               var;
        nNumberOfBytesToRead:   dword;
    var lpNumberOfBytesRead:    dword;
    var lpOverlapped:           OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadFile@20" );
```

Parameters

hFile

[in] Handle to the file to be read. The file handle must have been created with GENERIC_READ access to the file.

Windows NT/2000: 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** function.

Windows 95/98: For asynchronous read operations, *hFile* can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous read operations on mailslots, named pipes, or disk files.

lpBuffer

[out] Pointer to the buffer that receives the data read from the file.

nNumberOfBytesToRead

[in] Specifies the number of bytes to be read from the file.

lpNumberOfBytesRead

[out] Pointer to the variable that receives 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.

Windows NT/2000: 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**.

Windows 95/98: This parameter cannot be NULL.

lpOverlapped

[in] Pointer 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.

Windows NT/2000: 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.

Windows 95/98: For operations on files, disks, pipes, or mailslots, this parameter must be NULL; a pointer to an **OVERLAPPED** structure causes the call to fail. However, Windows 95/98 supports overlapped I/O on serial and parallel ports.

Return Values

The **ReadFile** function 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 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

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

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. To determine a volume's sector size, call the **GetDiskFreeSpace** function.

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 sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes.

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 time-outs as set and retrieved using the **SetCommTimeouts** and **GetCommTimeouts** functions. Unpredictable results can occur if you fail to set the time-out values. For more information about communication time-outs, 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
gOverLapped.Offset     = 0;
gOverLapped.OffsetHigh = 0;
gOverLapped.hEvent     = hEvent;

// 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
```

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CancelIo, CreateFile, GetCommTimeouts, GetOverlappedResult, GetQueued-CompletionStatus, OVERLAPPED, PeekNamedPipe, ReadFileEx, SetCommTimeouts, SetErrorMode, WriteFile

---

1.320 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 or canceled and the calling thread is in an alertable wait state.

```
ReadFileEx: procedure
(
        hFile:                  dword;
    var lpBuffer:               dword;
        nNumberOfBytesToRead:   dword;
    var lpOverlapped:           OVERLAPPED;
        lpCompletionRoutine:    procedure
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadFileEx@20" );
```

Parameters

hFile

[in] Handle to the file to be read. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and must have GENERIC_READ access to the file.

Windows NT/2000: This parameter 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** function.

Windows 95/98: This parameter can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot

perform asynchronous read operations on mailslots, named pipes, or disk files.

lpBuffer

[out] Pointer 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

[in] Specifies the number of bytes to be read from the file.

lpOverlapped

[in] Pointer 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.

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 queuing 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

[in] Pointer 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 input/output (I/O) 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

When using **ReadFileEx** you should check **GetLastError** even when the function returns "success" to check for conditions that are "successes" but have some outcome you might want to know about. For example, a buffer overflow when calling **ReadFileEx** will return TRUE, but **GetLastError** will report the overflow with ERROR_MORE_DATA. If the function call is successful and there are no warning conditions, **GetLastError** will return ERROR_SUCCESS.

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. To determine a volume's sector size, call the **GetDiskFreeSpace** function.

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 sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

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.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes.

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.

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 95/98: 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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CancelIo, CreateFile, FileIOCompletionRoutine, MsgWaitForMultipleObjectsEx, OVERLAPPED, ReadFile, SetErrorMode, SleepEx, WaitForMultipleObjectsEx, WaitForSingleObjectEx, WriteFileEx

---

1.321 ReadFileScatter

The **ReadFileScatter** function reads data from a file and stores the data into a set of buffers.

The function starts reading data from the file at a position specified by an **OVERLAPPED** structure. It operates asynchronously.

```
ReadFileScatter: procedure
(
        hFile:                  dword;
    var aSegmentArray:          FILE_SEGMENT_ELEMENT;
```

```
        nNumberOfBytesToRead:   dword;
    var lpReserved:             dword;
    var lpOverlapped:           OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadFileScatter@20" );
```

Parameters

hFile

[in] Handle to the file to be read.

This file handle must have been created using GENERIC_READ to specify read access to the file, FILE_FLAG_OVERLAPPED to specify asynchronous I/O, and FILE_FLAG_NO_BUFFERING to specify non-cached I/O.

aSegmentArray

[in] Pointer to an array of **FILE_SEGMENT_ELEMENT** pointers to buffers. The function stores the data it reads from the file into this set of buffers.

Each buffer must be at least the size of a system memory page and must be aligned on a system memory page size boundary. The system will read one system memory page of data into each buffer that a **FILE_SEGMENT_ELEMENT** pointer points to.

The function stores the data into the buffers in a sequential manner: it stores data into the first buffer, then into the second buffer, then into the next, filling each buffer, until there is no more data or there are no more buffers.

The final element of the array must be a 64-bit NULL pointer.

Note The array must contain one member for each system memory page-sized chunk of the total number of bytes to read from the file, plus one member for the final NULL pointer. For example, if the number of bytes to read is 40K, and the system page size is 4K, then this array must have 10 members for data, plus one member for the final NULL member, for a total of 11 members.

nNumberOfBytesToRead

[in] Specifies the total number of bytes to read from the file; each element of *aSegmentArray* contains a 1-page chunk of this total. Because the file must be opened with FILE_FLAG_NO_BUFFERING, the number of bytes to write must be a multiple of the sector size of the file system on which the file resides.

lpReserved

[in] This parameter is reserved for future use. You must set it to NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** data structure.

The **ReadFileScatter** function requires a valid **OVERLAPPED** structure. The *lpOverlapped* parameter cannot be NULL.

The **ReadFileScatter** function starts reading data from the file at a position specified by the **Offset** and **OffsetHigh** members of the **OVERLAPPED** structure.

The **ReadFileScatter** function may return before the read operation has completed. In that case, the **ReadFileScatter** function returns the value zero, and the **GetLastError** function returns the value ERROR_IO_PENDING. This asynchronous operation of **ReadFileScatter** lets the calling process continue while the read operation completes. You can call the **GetOverlappedResult, HasOverlappedIoCompleted,** or **GetQueuedCompletionStatus** function to obtain information about the completion of the read operation.

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 the **GetLastError** function.

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

If the function returns before the read operation has completed, the function returns zero, and **GetLastError** returns ERROR_IO_PENDING.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP2 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, GetOverlappedResult, GetQueuedCompletionStatus, HasOverlappedIoCompleted, OVERLAPPED, ReadFile, ReadFileEx, WriteFileGather

---

1.322 ReadProcessMemory

The **ReadProcessMemory** function reads data from an area of memory in a specified process. The entire area to be read must be accessible, or the operation fails.

```
ReadProcessMemory: procedure
(
        hProcess:               dword;
    var lpBaseAddress:          var;
    var lpBuffer:               var;
        nSize:                  dword;
    var lpNumberOfBytesRead:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReadProcessMemory@20" );
```

Parameters

hProcess

[in] Handle to the process whose memory is being read. The handle must have PROCESS_VM_READ access to the process.

lpBaseAddress

[in] Pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access. If this is the case, the function proceeds; otherwise, the function fails.

lpBuffer

[out] Pointer to a buffer that receives the contents from the address space of the specified process.

nSize

[in] Specifies the requested number of bytes to read from the specified process.

lpNumberOfBytesRead

[out] Pointer to a variable that receives the number of bytes transferred into the specified buffer. If *lpNumberOfBytesRead* is NULL, 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**.

The function fails if the requested read operation crosses into an area of the process that is inaccessible.

Remarks

ReadProcessMemory copies the data in the specified address range from the address space of the specified process into the specified buffer of the current process. Any process that has a handle with PROCESS_VM_READ access can call the function. The process whose address space is read is typically, but not necessarily, being debugged.

The entire area to be read must be accessible. If it is not, the function fails as noted previously.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, WriteProcessMemory

1.323 ReleaseMutex

The **ReleaseMutex** function releases ownership of the specified mutex object.

```
ReleaseMutex: procedure
(
    hMutex: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReleaseMutex@4" );
```

Parameters

hMutex

[in] Handle to the mutex object. The **CreateMutex** or **OpenMutex** function returns this handle.

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 **ReleaseMutex** function fails if the calling thread does not own the mutex object.

A thread gets ownership of a mutex by specifying a handle to the mutex in one of the wait functions. The thread that creates a mutex object can also get immediate ownership without using one of the wait functions. When the owning thread no longer needs to own the mutex object, it calls the **ReleaseMutex** function.

While a thread has ownership of a mutex, it can specify the same mutex in additional wait-function calls without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **ReleaseMutex** once for each time that the mutex satisfied a wait.

Example

For an example that uses **ReleaseMutex**, see Using Mutex Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateMutex

---

1.324 ReleaseSemaphore

The **ReleaseSemaphore** function increases the count of the specified semaphore object by a specified amount.

```
ReleaseSemaphore: procedure
(
        hSemaphore:         dword;
        lReleaseCount:      dword;
    var lpPreviousCount:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ReleaseSemaphore@12" );
```

Parameters

hSemaphore

[in] Handle to the semaphore object. The **CreateSemaphore** or **OpenSemaphore** function returns this handle.

Windows NT/2000: This handle must have SEMAPHORE_MODIFY_STATE access. For more information, see Synchronization Object Security and Access Rights.

lReleaseCount

[in] Specifies the amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount would cause the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns FALSE.

lpPreviousCount

[out] Pointer to a variable to receive the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The state of a semaphore object is signaled when its count is greater than zero and nonsignaled when its count is equal to zero. The process that calls the **CreateSemaphore** function specifies the semaphore's initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one.

Typically, an application uses a semaphore to limit the number of threads using a resource. Before a thread uses the

resource, it specifies the semaphore handle in a call to one of the wait functions. When the wait function returns, it decreases the semaphore's count by one. When the thread has finished using the resource, it calls **ReleaseSemaphore** to increase the semaphore's count by one.

Another use of **ReleaseSemaphore** is during an application's initialization. The application can create a semaphore with an initial count of zero. This sets the semaphore's state to nonsignaled and blocks all threads from accessing the protected resource. When the application finishes its initialization, it uses **ReleaseSemaphore** to increase the count to its maximum value, to permit normal access to the protected resource.

Example

For an example that uses **ReleaseSemaphore**, see Using Semaphore Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateSemaphore, OpenSemaphore

---

1.325 RemoveDirectory

The **RemoveDirectory** function deletes an existing empty directory.

```
RemoveDirectory: procedure
(
    lpPathName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__RemoveDirectoryA@4" );
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to be removed. The path must specify an empty directory, and the calling process must have delete access to the directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

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

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateDirectory

1.326 RequestWakeupLatency

The **RequestWakeupLatency** function specifies roughly how quickly the computer should enter the working state.

```
RequestWakeupLatency: procedure
(
    latency:    LATENCY_TIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__RequestWakeupLatency@4" );
```

Parameters

latency

[in] Specifies the latency requirement on the time is takes to wake the computer. This parameter can be one of the following values.

Value	Description
LT_LOWEST_LATENCY	PowerSystemSleeping1 state (equivalent to ACPI state S0 and APM state Working).
LT_DONT_CARE	Any latency (default).

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function will fail if the device does not support wake-up operations or if the system is entering the sleeping state.

Remarks

The system uses the wake-up latency requirement when choosing a sleeping state. The latency is not guaranteed, because wake-up time is determined by the hardware design of the particular computer.

To cancel a latency request, call **RequestWakeupLatency** with LT_DONT_CARE.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Power Management Overview, Power Management Functions

1.327 ResetEvent

The **ResetEvent** function sets the specified event object to the nonsignaled state.

```
ResetEvent: procedure
(
    hEvent: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ResetEvent@4" );
```

Parameters

hEvent

[in] Handle to the event object. The **CreateEvent** or **OpenEvent** function returns this handle.

Windows NT/2000: The handle must have EVENT_MODIFY_STATE access. For more information, see Synchronization Object Security and Access Rights.

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 state of an event object remains nonsignaled until it is explicitly set to signaled by the **SetEvent** or **PulseEvent** function. This nonsignaled state blocks the execution of any threads that have specified the event object in a call to one of the wait functions.

The **ResetEvent** function is used primarily for manual-reset event objects, which must be set explicitly to the nonsignaled state. Auto-reset event objects automatically change from signaled to nonsignaled after a single waiting thread is released.

Example

For an example that uses **ResetEvent**, see Using Event Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateEvent, OpenEvent, PulseEvent, SetEvent

1.328 ResumeThread

The **ResumeThread** function decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

```
ResumeThread: procedure
(
    hThread:    dword
);
    stdcall;
```

```
    returns( "eax" );
    external( "__imp__ResumeThread@4" );
```

Parameters

hThread

[in] Handle to the thread to be restarted.

Windows NT/2000: The handle must have THREAD_SUSPEND_RESUME access to the thread. For more information, see Thread Security and Access Rights.

Return Values

If the function succeeds, the return value is the thread's previous suspend count.

If the function fails, the return value is -1. To get extended error information, call **GetLastError**.

Remarks

The **ResumeThread** function checks the suspend count of the subject thread. If the suspend count is zero, the thread is not currently suspended. Otherwise, the subject thread's suspend count is decremented. If the resulting value is zero, then the execution of the subject thread is resumed.

If the return value is zero, the specified thread was not suspended. If the return value is 1, the specified thread was suspended but was restarted. If the return value is greater than 1, the specified thread is still suspended.

Note that while reporting debug events, all threads within the reporting process are frozen. Debuggers are expected to use the **SuspendThread** and **ResumeThread** functions to limit the set of threads that can execute within a process. By suspending all threads in a process except for the one reporting a debug event, it is possible to "single step" a single thread. The other threads are not released by a continue operation if they are suspended.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, OpenThread, SuspendThread

---

1.329 ScrollConsoleScreenBuffer

The **ScrollConsoleScreenBuffer** function moves a block of data in a screen buffer. The effects of the move can be limited by specifying a clipping rectangle, so the contents of the screen buffer outside the clipping rectangle are unchanged.

```
ScrollConsoleScreenBuffer: procedure
(
        hConsoleOutput:         dword;
    var lpScrollRectangle:      SMALL_RECT;
    var lpClipRectangle:        SMALL_RECT;
        dwDestinationOrigin:    COORD;
    var lpFill:                 CHAR_INFO
);
    stdcall;
    returns( "eax" );
    external( "__imp__ScrollConsoleScreenBufferA@20" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_WRITE access.

lpScrollRectangle

[in] Pointer to a **SMALL_RECT** structure whose members specify the upper-left and lower-right coordinates of the screen buffer rectangle to be moved.

lpClipRectangle

[in] Pointer to a **SMALL_RECT** structure whose members specify the upper-left and lower-right coordinates of the screen buffer rectangle that is affected by the scrolling. This pointer can be NULL.

dwDestinationOrigin

[in] Specifies the upper-left corner of the new location of the *lpScrollRectangle* contents.

lpFill

[in] Pointer to a **CHAR_INFO** structure that specifies the character and color attributes to be used in filling the cells within the intersection of *lpScrollRectangle* and *lpClipRectangle* that were left empty as a result of the move.

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

ScrollConsoleScreenBuffer copies the contents of a rectangular region of a screen buffer, specified by the *lpScrollRectangle* parameter, to another area of the screen buffer. The target rectangle has the same dimensions as the *lpScrollRectangle* rectangle with its upper-left corner at the coordinates specified by the *dwDestinationOrigin* parameter. Those parts of *lpScrollRectangle* that do not overlap with the target rectangle are filled in with the character and color attributes specified by the *lpFill* parameter.

The clipping rectangle applies to changes made in both the *lpScrollRectangle* rectangle and the target rectangle. For example, if the clipping rectangle does not include a region that would have been filled by the contents of *lpFill*, the original contents of the region are left unchanged.

If the scroll or target regions extend beyond the dimensions of the screen buffer, they are clipped. For example, if *lpScrollRectangle* is the region contained by (0,0) and (19,19) and *dwDestinationOrigin* is (10,15), the target rectangle is the region contained by (10,15) and (29,34). However, if the screen buffer is 50 characters wide and 30 characters high, the target rectangle is clipped to (10,15) and (29,29). Changes to the screen buffer are also clipped according to *lpClipRectangle*, if the parameter specifies a **SMALL_RECT** structure. If the clipping rectangle is specified as (0,0) and (49,19), only the changes that occur in that region of the screen buffer are made.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CHAR_INFO, SetConsoleCP, SetConsoleOutputCP, SetConsoleWindowInfo, SMALL_RECT

1.330 SearchPath

The **SearchPath** function searches for the specified file.

```
SearchPath: procedure
(
        lpPath:         string;
        lpFileName:     string;
        lpExtension:    string;
        nBufferLength:  dword;
    var lpBuffer:       var;
    var lpFilePart:     var
);
    stdcall;
    returns( "eax" );
    external( "__imp__SearchPathA@24" );
```

Parameters

lpPath

[in] Pointer to a null-terminated string that specifies the path to be searched for the file. If this parameter is NULL, the function searches for a matching file in the following directories in the following sequence:

The directory from which the application loaded.

The current directory.

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

Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.

Windows NT/2000: The 16-bit Windows system directory. There is no Win32 function that retrieves the path of this directory, but it is searched. The name of this directory is SYSTEM.

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

The directories that are listed in the PATH environment variable.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the file to search for.

lpExtension

[in] Pointer to a null-terminated string that specifies an extension to be added to the file name when searching for the file. The first character of the file name extension must be a period (.). The extension is added only if the specified file name does not end with an extension.

If a file name extension is not required or if the file name contains an extension, this parameter can be NULL.

nBufferLength

[in] Specifies the length, in **TCHARs**, of the buffer that receives the valid path and file name.

lpBuffer

[out] Pointer to the buffer that receives the path and file name of the file found.

lpFilePart

[out] Pointer to the variable that receives the address (within *lpBuffer*) of the last component of the valid path and file name, which is the address of the character immediately following the final backslash (\) in the path.

Return Values

If the function succeeds, the value returned is the length, in **TCHARs**, of the string copied to the buffer, not including the terminating null character. If the return value is greater than *nBufferLength*, the value returned is the size of the buffer required to hold the path.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, FindFirstFile, FindNextFile, GetSystemDirectory, GetWindowsDirectory

SetCommBreak

The **SetCommBreak** function suspends character transmission for a specified communications device and places the transmission line in a break state until the **ClearCommBreak** function is called.

```
SetCommBreak: procedure
(
    hFile:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCommBreak@4" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, ClearCommBreak, CreateFile

---

1.331 SetCommConfig

The **SetCommConfig** function sets the current configuration of a communications device.

```
SetCommConfig: procedure
(
        hCommDev:   dword;
```

```
    var lpCC:       COMMCONFIG;
        dwSize:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCommConfig@12" );
```

Parameters

hCommDev

[in] Handle to the open communications device.

lpCC

[in] Pointer to a **COMMCONFIG** structure.

dwSize

[in] Specifies the size, in bytes, of the structure pointed to by *lpCC*.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, GetCommConfig, COMMCONFIG

1.332 SetCommMask

The **SetCommMask** function specifies a set of events to be monitored for a communications device.

```
SetCommMask: procedure
(
    hFile:      dword;
    dwEvtMask:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCommMask@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

dwEvtMask

[in] Specifies the events to be enabled. A value of zero disables all events. This parameter can be one or more of the following values.

Value	Meaning
EV_BREAK	A break was detected on input.
EV_CTS	The CTS (clear-to-send) signal changed state.
EV_DSR	The DSR (data-set-ready) signal changed state.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state.
EV_RXCHAR	A character was received and placed in the input buffer.
EV_RXFLAG	The event character was received and placed in the input buffer. The event character is specified in the device's **DCB** structure, which is applied to a serial port by using the **SetCommState** function.
EV_TXEMPTY	The last character in the output buffer was sent.

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 **SetCommMask** function specifies the set of events that can be monitored for a particular communications resource. A handle to the communications resource can be specified in a call to the **WaitCommEvent** function, which waits for one of the events to occur. To get the current event mask of a communications resource, use the **GetCommMask** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, DCB, GetCommMask, SetCommState, WaitCommEvent

---

1.333 SetCommState

The **SetCommState** function configures a communications device according to the specifications in a device-control block (a **DCB** structure). The function reinitializes all hardware and control settings, but it does not empty output or input queues.

```
SetCommState: procedure
(
        hFile:  dword;
    var lpDCB:  DCB
);
    stdcall;
    returns( "eax" );
```

```
    external( "__imp__SetCommState@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpDCB

[in] Pointer to a **DCB** structure that contains the configuration information for the specified communications device.

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 **SetCommState** function uses a **DCB** structure to specify the desired configuration. The **GetCommState** function returns the current configuration.

To set only a few members of the **DCB** structure, you should modify a **DCB** structure that has been filled in by a call to **GetCommState**. This ensures that the other members of the **DCB** structure have appropriate values.

The **SetCommState** function fails if the **XonChar** member of the **DCB** structure is equal to the **XoffChar** member.

When **SetCommState** is used to configure the 8250, the following restrictions apply to the values for the **DCB** structure's **ByteSize** and **StopBits** members:

The number of data bits must be 5 to 8 bits.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, BuildCommDCB, CreateFile, DCB, GetCommState

---

1.334 SetCommTimeouts

The **SetCommTimeouts** function sets the time-out parameters for all read and write operations on a specified communications device.

```
SetCommTimeouts: procedure
(
        hFile:          dword;
    var lpCommTimeouts: COMMTIMEOUTS
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCommTimeouts@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpCommTimeouts

[in] Pointer to a **COMMTIMEOUTS** structure that contains the new time-out values.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, COMMTIMEOUTS, GetCommTimeouts, ReadFile, ReadFileEx, WriteFile, WriteFileEx

---

1.335 SetComputerName

The **SetComputerName** function stores a new NetBIOS name for the local computer. The name is stored in the registry and takes effect the next time the user restarts the computer.

If the local computer is a node in a cluster, **SetComputerName** sets NetBIOS name of the local computer, not that of the cluster.

Windows 2000: To set the DNS host name or the DNS domain name, call the **SetComputerNameEx** function.

```
SetComputerName: procedure
(
    lpComputerName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetComputerNameA@4" );
```

Parameters

lpComputerName

[in] Pointer to a null-terminated character string that specifies the name that will be the computer name the next time the computer is started. The name must not be longer than MAX_COMPUTERNAME_LENGTH characters.

Windows 95/98: If this string contains one or more characters that are outside the standard character set, those characters are coerced into standard characters.

Windows NT/2000: If this string contains one or more characters that are outside the standard character set, **SetComputerName** returns ERROR_INVALID_PARAMETER. It does not coerce the characters outside the standard set.

The standard character set includes letters, numbers, and the following symbols: ! @ # $ % ^ & ') (. - _ { } ~ .

Return Values

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

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

Remarks

Applications using this function must have administrator rights.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

System Information Overview, System Information Functions, GetComputerName, GetComputerNameEx, SetComputerNameEx

---

1.336 SetConsoleActiveScreenBuffer

The **SetConsoleActiveScreenBuffer** function sets the specified screen buffer to be the currently displayed console screen buffer.

```
SetConsoleActiveScreenBuffer: procedure
(
    hConsoleOutput: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleActiveScreenBuffer@4" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

A console can have multiple screen buffers. **SetConsoleActiveScreenBuffer** determines which one is displayed. You can write to an inactive screen buffer and then use **SetConsoleActiveScreenBuffer** to display the buffer's contents.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CreateConsoleScreenBuffer

1.337 SetConsoleCP

The **SetConsoleCP** function sets the input code page used by the console associated with the calling process. A console uses its input code page to translate keyboard input into the corresponding character value.

```
SetConsoleCP: procedure
(
    wCodePageID:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleCP@4" );
```

Parameters

wCodePageID

[in] Specifies the identifier of the code page to set. The identifiers of the code pages available on the local computer are stored in the registry under the following key.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

To determine a console's current input code page, use the **GetConsoleCP** function. To set and retrieve a console's output code page, use the **SetConsoleOutputCP** and **GetConsoleOutputCP** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleCP, GetConsoleOutputCP, SetConsoleOutputCP

1.338 SetConsoleCtrlHandler

The **SetConsoleCtrlHandler** function adds or removes an application-defined **HandlerRoutine** function from the list of handler functions for the calling process.

Windows NT/2000: If no handler function is specified, the function sets an inheritable attribute that determines whether the calling process ignores CTRL+C signals.

```
SetConsoleCtrlHandler: procedure
(
    HandlerRoutine: procedure;
    Add:            dword
);
    stdcall;
```

```
returns( "eax" );
external( "__imp__SetConsoleCtrlHandler@8" );
```

Parameters

HandlerRoutine

[in] Pointer to the application-defined **HandlerRoutine** function to add or remove.

Windows NT/2000: This parameter can be NULL.

Add

[in] Specifies whether to add or remove the function pointed to by the *HandlerRoutine* parameter from the handler list. If this parameter is TRUE, the handler is added; if it is FALSE, the handler is removed.

Windows NT/2000: If the *HandlerRoutine* parameter is NULL, a TRUE value causes the calling process to ignore CTRL+C input, and a FALSE value restores normal processing of CTRL+C input. This attribute of ignoring or processing CTRL+C is inherited by child processes.

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

Each console process has its own list of application-defined **HandlerRoutine** functions that handle CTRL+C and CTRL+BREAK signals. The handler functions also handle signals generated by the system when the user closes the console, logs off, or shuts down the system. Initially, the handler list for each process contains only a default handler function that calls the **ExitProcess** function. A console process adds or removes additional handler functions by calling the **SetConsoleCtrlHandler** function, which does not affect the list of handler functions for other processes. When a console process receives any of the control signals, its handler functions are called on a last-registered, first-called basis until one of the handlers returns TRUE. If none of the handlers returns TRUE, the default handler is called.

For console processes, the CTRL+C and CTRL+BREAK key combinations are typically treated as signals (CTRL_C_EVENT and CTRL_C_BREAK_EVENT). When a console window with the keyboard focus receives CTRL+C or CTRL+BREAK, the signal is typically passed to all processes sharing that console.

CTRL+BREAK is always treated as a signal, but typical CTRL+C behavior can be changed in three ways that prevent the handler functions from being called:

The **SetConsoleMode** function can disable the **ENABLE_PROCESSED_INPUT** mode for a console's input buffer, so CTRL+C is reported as keyboard input rather than as a signal.

Windows NT/2000: Calling **SetConsoleCtrlHandler** with the NULL and TRUE arguments causes the calling process to ignore CTRL+C signals. This attribute is inherited by child processes, but it can be enabled or disabled by any process without affecting existing processes.

If a console process is being debugged and CTRL+C signals have not been disabled, the system generates a DBG_CONTROL_C exception. This exception is raised only for the benefit of the debugger, and an application should never use an exception handler to deal with it. If the debugger handles the exception, an application will not notice the CTRL+C, with one exception: alertable waits will terminate. If the debugger passes the exception on unhandled, CTRL+C is passed to the console process and treated as a signal, as previously discussed.

A console process can use the **GenerateConsoleCtrlEvent** function to send a CTRL+C or CTRL+BREAK signal to a console process group.

The system generates CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT, and CTRL_SHUTDOWN_EVENT signals when the user closes the console, logs off, or shuts down the system so that the process has an opportunity to clean up before termination. Console functions, or any C run-time functions that call console functions, may not work reliably during processing of any of the three signals mentioned previously. The reason is that some or all of the internal console cleanup routines may have been called before executing the process signal handler.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, ExitProcess, GenerateConsoleCtrlEvent, GetConsoleMode, HandlerRoutine, SetConsoleMode

---

1.339 SetConsoleCursorInfo

The **SetConsoleCursorInfo** function sets the size and visibility of the cursor for the specified console screen buffer.

```
SetConsoleCursorInfo: procedure
(
        hConsoleOutput:         dword;
    var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleCursorInfo@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_WRITE access.

lpConsoleCursorInfo

[in] Pointer to a **CONSOLE_CURSOR_INFO** structure containing the new specifications for the screen buffer's cursor.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

When a screen buffer's cursor is visible, its appearance can vary, ranging from completely filling a character cell to showing up as a horizontal line at the bottom of the cell. The **dwSize** member of the **CONSOLE_CURSOR_INFO** structure specifies the percentage of a character cell that is filled by the cursor. If this member is less than 1 or greater than 100, **SetConsoleCursorInfo** fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Wincon.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CONSOLE_CURSOR_INFO, GetConsoleCursorInfo, SetConsoleCursorPosition

1.340 SetConsoleCursorPosition

The **SetConsoleCursorPosition** function sets the cursor position in the specified console screen buffer.

```
SetConsoleCursorPosition: procedure
(
    hConsoleOutput:     dword;
    dwCursorPosition:   COORD
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleCursorPosition@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_WRITE access.

dwCursorPosition

[in] Specifies a **COORD** structure containing the new cursor position. The coordinates are the column and row of a screen buffer character cell. The coordinates must be within the boundaries of the screen buffer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The cursor position determines where characters written by the **WriteFile** or **WriteConsole** function, or echoed by the **ReadFile** or **ReadConsole** function, are displayed. To determine the current position of the cursor, use the **GetConsoleScreenBufferInfo** function.

If the new cursor position is not within the boundaries of the screen buffer's window, the window origin changes to make the cursor visible.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleCursorInfo, GetConsoleScreenBufferInfo, ReadConsole, ReadFile, SetConsoleCursorInfo, WriteConsole, WriteFile

1.341 SetConsoleMode

The **SetConsoleMode** function sets the input mode of a console's input buffer or the output mode of a console screen buffer.

```
SetConsoleMode: procedure
(
    hConsoleHandle: dword;
    dwMode:         dword
);
```

```
stdcall;
returns( "eax" );
external( "__imp__SetConsoleMode@8" );
```

Parameters

hConsoleHandle

[in] Handle to a console input buffer or a screen buffer. The handle must have GENERIC_WRITE access.

dwMode

[in] Specifies the input or output mode to set. If the *hConsoleHandle* parameter is an input handle, the mode can be a combination of the following values. When a console is created, all input modes except ENABLE_WINDOW_INPUT are enabled by default.

Value	Meaning
ENABLE_LINE_INPUT	The **ReadFile** or **ReadConsole** function returns only when a carriage return character is read. If this mode is disabled, the functions return when one or more characters are available.
ENABLE_ECHO_INPUT	Characters read by the **ReadFile** or **ReadConsole** function are written to the active screen buffer as they are read. This mode can be used only if the ENABLE_LINE_INPUT mode is also enabled.
ENABLE_PROCESSED_INPUT	CTRL+C is processed by the system and is not placed in the input buffer. If the input buffer is being read by **ReadFile** or **ReadConsole**, other control keys are processed by the system and are not returned in the **ReadFile** or **ReadConsole** buffer. If the ENABLE_LINE_INPUT mode is also enabled, backspace, carriage return, and linefeed characters are handled by the system.
ENABLE_WINDOW_INPUT	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer by applications using the **ReadConsoleInput** function, but not by those using **ReadFile** or **ReadConsole**.
ENABLE_MOUSE_INPUT	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. These events are discarded by **ReadFile** or **ReadConsole**, even when this mode is enabled.

If the *hConsoleHandle* parameter is a screen buffer handle, the mode can be a combination of the following values. When a screen buffer is created, both output modes are enabled by default.

Value	Meaning
ENABLE_PROCESSED_OUTPUT	Characters written by the **WriteFile** or **WriteConsole** function or echoed by the **ReadFile** or **ReadConsole** function are examined for ASCII control sequences and the correct action is performed. Backspace, tab, bell, carriage return, and linefeed characters are processed.

ENABLE_WRAP_AT_EOL_OUTPUT	When writing with **WriteFile** or **WriteConsole** or echoing with **ReadFile** or **ReadConsole**, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the screen buffer to scroll up (discarding the top row of the screen buffer) when the cursor advances beyond the last row in the screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

A console consists of an input buffer and one or more screen buffers. The mode of a console buffer determines how the console behaves during input and output (I/O) operations. One set of flag constants is used with input handles, and another set is used with screen buffer (output) handles. Setting the output modes of one screen buffer does not affect the output modes of other screen buffers.

The ENABLE_LINE_INPUT and ENABLE_ECHO_INPUT modes only affect processes that use **ReadFile** or **ReadConsole** to read from the console's input buffer. Similarly, the ENABLE_PROCESSED_INPUT mode primarily affects **ReadFile** and **ReadConsole** users, except that it also determines whether Ctrl+C input is reported in the input buffer (to be read by the **ReadConsoleInput** function) or is passed to a **HandlerRoutine** function defined by the application.

The ENABLE_WINDOW_INPUT and ENABLE_MOUSE_INPUT modes determine whether user interactions involving window resizing and mouse actions are reported in the input buffer or discarded. These events can be read by **ReadConsoleInput**, but they are always filtered by **ReadFile** and **ReadConsole**.

The ENABLE_PROCESSED_OUTPUT and ENABLE_WRAP_AT_EOL_OUTPUT modes only affect processes using **ReadFile** or **ReadConsole** and **WriteFile** or **WriteConsole**.

To determine the current mode of a console input buffer or a screen buffer, use the **GetConsoleMode** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleMode, HandlerRoutine, ReadConsole, ReadConsoleInput, ReadFile, WriteConsole, WriteFile

---

1.342 SetConsoleOutputCP

The **SetConsoleOutputCP** function sets the output code page used by the console associated with the calling process. A console uses its output code page to translate the character values written by the various output functions into the images displayed in the console window.

```
SetConsoleOutputCP: procedure
(
    wCodePageID:    dword
);
```

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

Parameters

wCodePageID

[in] Specifies the identifier of the code page to set. The identifiers of the code pages available on the local computer are stored in the registry under the following key.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

To determine a console's current output code page, use the **GetConsoleOutputCP** function. To set and retrieve a console's input code page, use the **SetConsoleCP** and **GetConsoleCP** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleCP, GetConsoleOutputCP, SetConsoleCP

1.343 SetConsoleScreenBufferSize

The **SetConsoleScreenBufferSize** function changes the size of the specified console screen buffer.

```
SetConsoleScreenBufferSize: procedure
(
    hConsoleOutput: dword;
    dwSize:         COORD
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleScreenBufferSize@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_WRITE access.

dwSize

[in] Specifies a **COORD** structure containing the new size, in rows and columns, of the screen buffer. The specified

width and height cannot be less than the width and height of the screen buffer's window. The specified dimensions also cannot be less than the minimum size allowed by the system. This minimum depends on the current font size for the console (selected by the user) and the SM_CXMIN and SM_CYMIN values returned by the **GetSystemMetrics** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, COORD, GetConsoleScreenBufferInfo, SetConsoleWindowInfo

---

1.344 SetConsoleTextAttribute

The **SetConsoleTextAttribute** function sets the foreground (text) and background color attributes of characters written to the screen buffer by the **WriteFile** or **WriteConsole** function, or echoed by the **ReadFile** or **ReadConsole** function. This function affects only text written after the function call.

```
SetConsoleTextAttribute: procedure
(
    hConsoleOutput: dword;
    wAttributes:    word
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleTextAttribute@8" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_READ access.

wAttributes

[in] Specifies the foreground and background color attributes. Any combination of the following values can be specified: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces white text on a black background:

```
FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE
```

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 determine the current color attributes of a screen buffer, call the **GetConsoleScreenBufferInfo** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleScreenBufferInfo, ReadConsole, ReadFile, WriteConsole, WriteFile

---

1.345 SetConsoleTitle

The **SetConsoleTitle** function sets the title bar string for the current console window.

```
SetConsoleTitle: procedure
(
    lpConsoleTitle: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleTitleA@4" );
```

Parameters

lpConsoleTitle

[in] Pointer to a null-terminated string that contains the string to appear in the title bar of the console window.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **`SetConsoleCP`** or **`SetConsoleOutputCP`** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleTitle, SetConsoleCP, SetConsoleOutputCP

1.346 SetConsoleWindowInfo

The **SetConsoleWindowInfo** function sets the current size and position of a console screen buffer's window.

```
SetConsoleWindowInfo: procedure
(
        hConsoleOutput:     dword;
        bAbsolute:          dword;
    var lpConsoleWindow:    SMALL_RECT
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetConsoleWindowInfo@12" );
```

Parameters

hConsoleOutput

[in] Handle to a console screen buffer. The handle must have GENERIC_WRITE access.

bAbsolute

[in] Specifies how the coordinates in the structure pointed to by the *lpConsoleWindow* parameter are used. If *bAbsolute* is TRUE, the coordinates specify the new upper-left and lower-right corners of the window. If it is FALSE, the coordinates are offsets to the current window-corner coordinates.

lpConsoleWindow

[in] Pointer to a **SMALL_RECT** structure that contains values that determine the new upper-left and lower-right corners of the window.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The function fails if the specified window rectangle extends beyond the boundaries of the screen buffer. This means that the **Top** and **Left** members of the *lpConsoleWindow* rectangle (or the calculated top and left coordinates, if *bAbsolute* is FALSE) cannot be less than zero. Similarly, the **Bottom** and **Right** members (or the calculated bottom and right coordinates) cannot be greater than (screen buffer height – 1) and (screen buffer width – 1), respectively. The function also fails if the **Right** member (or calculated right coordinate) is less than or equal to the **Left** member (or calculated left coordinate) or if the **Bottom** member (or calculated bottom coordinate) is less than or equal to the **Top** member (or calculated top coordinate).

For consoles with more than one screen buffer, changing the window location for one screen buffer does not affect the window locations of the other screen buffers.

To determine the current size and position of a screen buffer's window, use the **GetConsoleScreenBufferInfo** function. This function also returns the maximum size of the window, given the current screen buffer size, the current font size, and the screen size. The **GetLargestConsoleWindowSize** function returns the maximum window size given the current font and screen sizes, but it does not consider the size of the screen buffer.

SetConsoleWindowInfo can be used to scroll the contents of the screen buffer by shifting the position of the window rectangle without changing its size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleScreenBufferInfo, GetLargestConsoleWindowSize, SMALL_RECT, ScrollConsoleScreenBuffer

---

1.347 SetCriticalSectionSpinCount

The **SetCriticalSectionSpinCount** function sets the spin count for the specified critical section.

```
SetCriticalSectionSpinCount: procedure
(
    var lpCriticalSection:  CRITICAL_SECTION;
        dwSpinCount:        dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCriticalSectionSpinCount@8" );
```

Parameters

lpCriticalSection

[in/out] Pointer to the critical section object.

dwSpinCount

[in] Specifies the spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin *dwSpinCount* times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Return Values

The function returns the previous spin count for the critical section.

Remarks

The threads of a single process can use a critical section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function to initialize the object. You can subsequently modify the spin count by calling the **SetCriticalSectionSpinCount** function.

The spin count is useful for critical sections of short duration that can experience high levels of contention. Consider a worst-case scenario, in which an application on an SMP system has two or three threads constantly allocating and releasing memory from the heap. The application serializes the heap with a critical section. In the worst-case scenario, contention for the critical section is constant, and each thread makes an expensive call to the **WaitForSingleObject** function. However, if the spin count is set properly, the calling thread will not immediately call **WaitForSingleObject** when contention occurs. Instead, the calling thread can acquire ownership of the critical section if it is released during the spin operation.

You can improve performance significantly by choosing a small spin count for a critical section of short duration. The heap manager uses a spin count of roughly 4000 for its per-heap critical sections. This gives great performance and scalability in almost all worst-case scenarios.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Unsupported.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, WaitForSingleObject

---

1.348 SetCurrentDirectory

The **SetCurrentDirectory** function changes the current directory for the current process.

```
SetCurrentDirectory: procedure
(
    lpPathName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetCurrentDirectoryA@4" );
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path to the new current directory. This parameter may be a relative path or a full path. In either case, the full path of the specified directory is calculated and stored as the current directory.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

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

Each process has a single current directory made up of two parts:

A disk designator that is either a drive letter followed by a colon, or a server name and share name (*servername**sharename*)

A directory on the disk designator

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetCurrentDirectory

1.349 SetDefaultCommConfig

The **SetDefaultCommConfig** function sets the default configuration for a communications device.

```
SetDefaultCommConfig: procedure
(
        lpszName:   string;
    var lpCC:       COMMCONFIG;
        dwSize:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetDefaultCommConfigA@12" );
```

Parameters

lpszName

[in] Pointer to a null-terminated string specifying the name of the device.

lpCC

[in] Pointer to a **COMMCONFIG** structure.

dwSize

[in] Specifies the size, in bytes, of the structure pointed to by *lpCC*.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, GetDefaultCommConfig, COMMCONFIG

SetEndOfFile

The **SetEndOfFile** function moves the end-of-file (EOF) position for the specified file to the current position of the file pointer.

```
SetEndOfFile: procedure
(
    hFile:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetEndOfFile@4" );
```

Parameters

hFile

[in] Handle to the file to have its EOF position moved. The file handle must have been created with GENERIC_WRITE access to the file.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

This function can be used to truncate or extend a file. If the file is extended, the contents of the file between the old EOF position and the new position are not defined.

If you called **CreateFileMapping** to create a file-mapping object for *hFile*, you must first call **UnmapViewOfFile** to unmap all views and call **CloseHandle** to close the file-mapping object before you can call **SetEndOfFile**.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CloseHandle, CreateFile, CreateFileMapping, UnmapViewOfFile

---

1.350 SetEnvironmentVariable

The **SetEnvironmentVariable** function sets the value of an environment variable for the current process.

```
SetEnvironmentVariable: procedure
(
    lpName:     string;
    lpValue:    string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetEnvironmentVariableA@8" );
```

Parameters

lpName

[in] Pointer to a null-terminated string that specifies the environment variable whose value is being set. The operating system creates the environment variable if it does not exist and *lpValue* is not NULL.

lpValue

[in] Pointer to a null-terminated string containing the new value of the specified environment variable. If this parameter is NULL, the variable is deleted from the current process's environment.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

This function has no effect on the system environment variables or the environment variables of other processes.

To add or modify system environment variables, the user selects **System** from the **Control Panel**, then selects the **Environment** tab. The user can also add or modify environment variables at a command prompt using the **set** command. Environment variables created with the **set** command apply only to the command window in which they are set, and to its child processes. For more information, type **set /?** at a command prompt.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetEnvironmentVariable

---

1.351 SetErrorMode

The **SetErrorMode** function controls whether the system will handle the specified types of serious errors, or whether the process will handle them.

```
SetErrorMode: procedure
(
    uMode:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetErrorMode@4" );
```

Parameters

uMode

[in] Specifies the process error mode. This parameter can be one or more of the following values.

Value	Action
0	Use the system default, which is to display all error dialog boxes.
SEM_FAILCRITICALERRORS	The system does not display the critical-error-handler message box. Instead, the system sends the error to the calling process.
SEM_NOALIGNMENTFAULTEXCEPT	**Itanium or RISC:** The system automatically fixes memory alignment faults and makes them invisible to the application. It does this for the calling process and any descendant processes. This value has no effect on x86 processors.
SEM_NOGPFAULTERRORBOX	The system does not display the general-protection-fault message box. This flag should *only* be set by debugging applications that handle general protection (GP) faults themselves with an exception handler.
SEM_NOOPENFILEERRORBOX	The system does not display a message box when it fails to find a file. Instead, the error is returned to the calling process.

Return Values

The return value is the previous state of the error-mode bit flags.

Remarks

Each process has an associated error mode that indicates to the system how the application is going to respond to serious errors. A child process inherits the error mode of its parent process.

Itanium: An application must explicitly call **SetErrorMode** with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults. The default setting is for the system to make alignment faults visible to an application

x86: The system does not make alignment faults visible to an application. Therefore, specifying SEM_NOALIGNMENTFAULTEXCEPT is not an error, but the system is free to silently ignore the request. This means that code sequences such as the following are not always valid on x86 computers:

```
SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT);

fuOldErrorMode = SetErrorMode(0);

ASSERT(fuOldErrorMode == SEM_NOALIGNMENTFAULTEXCEPT);
```

RISC: Misaligned memory references cause an alignment fault exception. To control whether the system automatically fixes such alignment faults or makes them visible to an application, use SEM_NOALIGNMENTFAULTEXCEPT.

MIPS: An application must explicitly call **SetErrorMode** with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults. The default setting is for the system to make alignment faults visible to an application.

Alpha: To control the alignment fault behavior, set the **EnableAlignmentFaultExceptions** value in the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager** registry key as follows.

Value	Meaning
0	Automatically fix alignment faults. This is the default.
1	Make alignment faults visible to the application. You must call **SetErrorMode** with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions

1.352 SetEvent

The **SetEvent** function sets the specified event object to the signaled state.

```
SetEvent: procedure
(
    hEvent: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetEvent@4" );
```

Parameters

hEvent

[in] Handle to the event object. The **CreateEvent** or **OpenEvent** function returns this handle.

Windows NT/2000: The handle must have EVENT_MODIFY_STATE access. For more information, see Synchronization Object Security and Access Rights.

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 state of a manual-reset event object remains signaled until it is set explicitly to the nonsignaled state by the **ResetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object by calling one of the wait functions, can be released while the object's state is signaled.

The state of an auto-reset event object remains signaled until a single waiting thread is released, at which time the system automatically sets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled.

Example

For an example that uses **SetEvent**, see Using Event Objects.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CreateEvent, OpenEvent, PulseEvent, ResetEvent

---

1.353 SetFileApisToANSI

The **SetFileApisToANSI** function causes the file I/O functions to use the ANSI character set code page. This function is useful for 8-bit console input and output operations.

```
SetFileApisToANSI: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__SetFileApisToANSI@0" );
```

Parameters

This function has no parameters.

Return Values

This function has no return value.

Remarks

The file I/O functions whose code page is set by **SetFileApisToANSI** are those functions exported by KERNEL32.DLL that accept or return a file name. **SetFileApisToANSI** sets the code page per process, rather than per thread or per computer.

The **SetFileApisToANSI** function complements the **SetFileApisToOEM** function, which causes the same set of file

I/O functions to use the OEM character set code page.

The 8-bit console functions use the OEM code page by default. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by other functions, and vice versa. For example, if the **FindFirstFileA** function returns a string that contains certain extended ANSI characters, and the 8-bit console functions are set to use the OEM code page, then the **WriteConsoleA** function does not display the string properly.

Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. Use the **SetConsoleCP** and **SetConsoleOutputCP** functions to set the code page for the 8-bit console functions.

To solve the problem of code page incompatibility, it is best to use Unicode for console applications. Console applications that use Unicode are much more versatile than those that use 8-bit console functions. Barring that solution, a console application can call the **SetFileApisToOEM** function to cause the set of file I/O functions to use OEM character set strings rather than ANSI character set strings. Use the **SetFileApisToANSI** function to set those functions back to the ANSI code page.

When dealing with command lines, a console application should obtain the command line in Unicode form and then convert it to OEM form using the relevant character-to-OEM functions. Note also that the array in the *argv* parameter of the command-line **main** function contains ANSI character set strings in this case.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, AreFileApisANSI, FindFirstFileA, SetFileApisToOEM, SetConsoleCP, SetConsoleOutputCP, WriteConsoleA

1.354 SetFileApisToOEM

The **SetFileApisToOEM** function causes the file I/O functions to use the OEM character set code page. This function is useful for 8-bit console input and output operations.

```
SetFileApisToOEM: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__SetFileApisToOEM@0" );
```

Parameters

This function has no parameters.

Return Values

This function has no return value.

Remarks

The file I/O functions whose code page is set by **SetFileApisToOEM** are those functions exported by KERNEL32.DLL that accept or return a file name. **SetFileApisToOEM** sets the code page per process, rather than per thread or per computer.

The **SetFileApisToOEM** function is complemented by the **SetFileApisToANSI** function, which causes the same set of file I/O functions to use the ANSI character set code page.

The 8-bit console functions use the OEM code page by default. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by other functions,

and vice versa. For example, if the **FindFirstFileA** function returns a string that contains certain extended ANSI characters, and the 8-bit console functions are set to use the OEM code page, then the **WriteConsoleA** function will not display the string properly.

Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. Use the **SetConsoleCP** and **SetConsoleOutputCP** functions to set the code page for the 8-bit console functions.

To solve the problem of code page incompatibility, it is best to use Unicode for console applications. Console applications that use Unicode are much more versatile than those that use 8-bit console functions. Barring that solution, a console application can call the **SetFileApisToOEM** function to cause the set of file I/O functions to use OEM character set strings rather than ANSI character set strings. Use the **SetFileApisToANSI** function to set those functions back to the ANSI code page.

When dealing with command lines, a console application should obtain the command line in Unicode form and then convert it to OEM form using the relevant character-to-OEM functions. Note also that the array in the *argv* parameter of the command-line **main** function contains ANSI character set strings in this case.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, AreFileApisANSI, FindFirstFileA, SetConsoleCP, SetConsoleCP, SetConsoleOutputCP, SetFileApisToANSI, WriteConsoleA

1.355 SetFileAttributes

The **SetFileAttributes** function sets a file's attributes.

```
SetFileAttributes: procedure
(
    lpFileName:         string;
    dwFileAttributes:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetFileAttributesA@8" );
```

Parameters

lpFileName

[in] Pointer to a string that specifies the name of the file whose attributes are to be set.

Windows NT/2000: In the ANSI version of this function, the name is limited to MAX_PATH characters. To extend this limit to nearly 32,000 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see File Name Conventions.

Windows 95/98: This string must not exceed MAX_PATH characters.

dwFileAttributes

[in] Specifies the file attributes to set for the file. This parameter can be one or more of the following values. However, all other values override FILE_ATTRIBUTE_NORMAL.

Attribute	Meaning

FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this attribute 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 attribute is valid only if used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
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

The following table describes how to set the attributes that cannot be set using **SetFileAttributes**.

Attribute	How to Set
FILE_ATTRIBUTE_COMPRESSED	To set a file's compression state, use the **`DeviceIoControl`** function with the **FSCTL_SET_COMPRESSION** operation.
FILE_ATTRIBUTE_DEVICE	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY	Files cannot be converted into directories. To create a directory, use the **`CreateDirectory`** or **`CreateDirectoryEx`** function.
FILE_ATTRIBUTE_ENCRYPTED	To create an encrypted file, use the **CreateFile** function with the FILE_ATTRIBUTE_ENCRYPTED attribute. To convert an existing file into an encrypted file, use the **`EncryptFile`** function.
FILE_ATTRIBUTE_REPARSE_POINT	To associate a reparse point with a file, use the **`DeviceIoControl`** function with the **FSCTL_SET_REPARSE_POINT** operation.
FILE_ATTRIBUTE_SPARSE_FILE	To set a file's sparse attribute, use the **`DeviceIoControl`** function with the **FSCTL_SET_SPARSE** operation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, File I/O Functions, GetFileAttributes

1.356 SetFilePointer

The **SetFilePointer** function moves the file pointer of an open file.

This function stores the file pointer in two **DWORD** values. To more easily work with file pointers that are larger than a single **DWORD** value, use the **SetFilePointerEx** function.

```
SetFilePointer: procedure
(
        hFile:                  dword;
        lDistanceToMove:        dword;
    var lpDistanceToMoveHigh:   dword;
        dwMoveMethod:           dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetFilePointer@16" );
```

Parameters

hFile

[in] Handle to the file whose file pointer is to be moved. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file.

lDistanceToMove

[in] Low-order 32 bits of a signed value that specifies the number of bytes to move the file pointer. If *lpDistanceToMoveHigh* is not NULL, *lpDistanceToMoveHigh* and *lDistanceToMove* form a single 64-bit signed value that specifies the distance to move. If *lpDistanceToMoveHigh* is NULL, *lDistanceToMove* is a 32-bit signed value. A positive value for *lDistanceToMove* moves the file pointer forward in the file, and a negative value moves the file pointer backward.

lpDistanceToMoveHigh

[in] Pointer to the high-order 32 bits of the signed 64-bit distance to move. If you do not need the high-order 32 bits, this pointer may be NULL. When non-NULL, this parameter also receives the high-order **DWORD** of the new value of the file pointer. For more information, see the Remarks section later in this topic.

Windows 95/98: If the pointer *lpDistanceToMoveHigh* is not NULL, then it must point to either 0, INVALID_SET_FILE_POINTER, or the sign extension of the value of *lDistanceToMove*. Any other value will be rejected.

dwMoveMethod

[in] Starting point for the file pointer move. This parameter can be one of the following values.

Value	Meaning

FILE_BEGIN	The starting point is zero or the beginning of the file.
FILE_CURRENT	The starting point is the current value of the file pointer.
FILE_END	The starting point is the current end-of-file position.

Return Values

If the **SetFilePointer** function succeeds and *lpDistanceToMoveHigh* is NULL, the return value is the low-order **DWORD** of the new file pointer. If *lpDistanceToMoveHigh* is not NULL, the function returns the low order **DWORD** of the new file pointer, and puts the high-order **DWORD** of the new file pointer into the **LONG** pointed to by that parameter.

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

If the function fails, and *lpDistanceToMoveHigh* is non-NULL, the return value is INVALID_SET_FILE_POINTER. However, because INVALID_SET_FILE_POINTER is a valid value for the low-order **DWORD** of the new file pointer, you must check **GetLastError** to determine whether an error occurred. If an error occurred, **GetLastError** returns a value other than NO_ERROR. For a code example that illustrates this point, see the Remarks section later in this topic.

If the new file pointer would have been a negative value, the function fails, the file pointer is not moved, and the code returned by **GetLastError** is ERROR_NEGATIVE_SEEK.

Remarks

You cannot use the **SetFilePointer** function with a handle to a nonseeking device such as a pipe or a communications device. To determine the file type for *hFile*, use the **GetFileType** function.

To determine the present position of a file pointer, see Retrieving a File Pointer.

Use caution when setting the file pointer in a multithreaded application. You must synchronize access to shared resources. For example, an application whose threads share a file handle, update the file pointer, and read from the file must protect this sequence by using a critical section object or mutex object. For more information about these objects, see Critical Section Objects and Mutex Objects.

If the *hFile* file handle was opened with the FILE_FLAG_NO_BUFFERING flag set, an application can move the file pointer only to sector-aligned positions. A *sector-aligned position* is a position that is a whole number multiple of the volume's sector size. An application can obtain a volume's sector size by calling the**GetDiskFreeSpace** function. If an application calls **SetFilePointer** with distance-to-move values that result in a position that is not sector-aligned and a handle that was opened with FILE_FLAG_NO_BUFFERING, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

Note that it is not an error to set the file pointer to a position beyond the end of the file. The size of the file does not increase until you call the **SetEndOfFile, WriteFile**, or **WriteFileEx** function. A write operation increases the size of the file to the file pointer position plus the size of the buffer written, leaving the intervening bytes uninitialized.

If the return value is INVALID_SET_FILE_POINTER and if *lpDistanceToMoveHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed. The following sample code illustrates this point:

```
// Case One: calling the function with lpDistanceToMoveHigh == NULL

// Try to move hFile's file pointer some distance.
dwPtr = SetFilePointer (hFile, lDistance, NULL, FILE_BEGIN) ;

if (dwPtr == INVALID_SET_FILE_POINTER) // Test for failure
{
    // Obtain the error code.
    dwError = GetLastError() ;

    // Deal with failure.
    // . . .
```

```
} // End of error handler


//
// Case Two: calling the function with lpDistanceToMoveHigh != NULL

// Try to move hFile's file pointer some huge distance.
dwPtrLow = SetFilePointer (hFile, lDistLow, & lDistHigh, FILE_BEGIN) ;

// Test for failure
if (dwPtrLow == INVALID_SET_FILE_POINTER && (dwError = GetLastError()) != NO_ERROR )
{
    // Deal with failure.
    // . . .

} // End of error handler
```

The parameter *lpDistanceToMoveHigh* is used to manipulate huge files. If it is set to NULL, then *lDistanceToMove* has a maximum value of 2^31—2, or 2 gigabytes less two. This is because all file pointer values are signed values. Therefore if there is even a small chance that the file will grow to that size, you should treat the file as a huge file and work with 64-bit file pointers. With file compression on NTFS, and sparse files, it is possible to have files that are large even if the underlying volume is not very large.

If *lpDistanceToMoveHigh* is not NULL, then *lpDistanceToMoveHigh* and *lDistanceToMove* form a single 64-bit signed value. The *lDistanceToMove* parameter is treated as the low-order 32 bits of the value, and *lpDistanceToMove-High* as the upper 32 bits. Thus, *lpDistanceToMoveHigh* is a sign extension of *lDistanceToMove.*

To move the file pointer from zero to 2 gigabytes, *lpDistanceToMoveHigh* can be either NULL or a sign extension of *lDistanceToMove.* To move the pointer more than 2 gigabytes, use *lpDistanceToMoveHigh* and *lDistanceToMove* as a single 64-bit quantity. For example, to move in the range from 2 gigabytes to 4 gigabytes set the contents of *lpDistanceToMoveHigh* to zero, or to –1 for a negative sign extension of *lDistanceToMove.*

To work with 64-bit file pointers, you can declare a **LONG**, treat it as the upper half of the 64-bit file pointer, and pass its address in *lpDistanceToMoveHigh.* This means you have to treat two different variables as a logical unit, which is error-prone. The problems can be ameliorated by using the **LARGE_INTEGER** structure to create a 64-bit value and passing the two 32-bit values by means of the appropriate elements of the union.

It is conceptually simpler and better design to use a function to hide the interface to **SetFilePointer**. To do so, use something like this:

```
__int64 myFileSeek (HANDLE hf, __int64 distance, DWORD MoveMethod)
{
   LARGE_INTEGER li;

   li.QuadPart = distance;

   li.LowPart = SetFilePointer (hf, li.LowPart, &li.HighPart, MoveMethod);

   if (li.LowPart == INVALID_SET_FILE_POINTER && GetLastError() != NO_ERROR)
   {
      li.QuadPart = -1;
   }

   return li.QuadPart;
}
```

Note You can use **SetFilePointer** to determine the length of a file. To do this, use FILE_END for *dwMoveMethod* and seek to location zero. The file offset returned is the length of the file. However, this practice can have unintended side effects, such as failure to save the current file pointer so that the program can return to that location. It is simpler and safer to use **`GetFileSize`** instead.

You can also use the **SetFilePointer** function to query the current file pointer position. To do this, specify a move method of FILE_CURRENT and a distance of zero.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetDiskFreeSpace, GetFileSize, GetFileType, ReadFile, ReadFileEx, SetEndOfFile, SetFilePointerEx, WriteFile, WriteFileEx

---

1.357 SetFileTime

The **SetFileTime** function sets the date and time that a file was created, last accessed, or last modified.

```
SetFileTime: procedure
(
        hFile:              dword;
    var lpCreationTime:     FILETIME;
    var lpLastAccessTime:   FILETIME;
    var lpLastWriteTime:    FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetFileTime@16" );
```

Parameters

hFile

[in] Handle to the file for which to set the dates and times. The file handle must have been created with GENERIC_WRITE access to the file.

lpCreationTime

[in] Pointer to a **FILETIME** structure that contains the date and time the file was created. This parameter can be NULL if the application does not need to set this information.

lpLastAccessTime

[in] Pointer to a **FILETIME** structure that contains 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 need to set this information.

lpLastWriteTime

[in] Pointer to a **FILETIME** structure that contains the date and time the file was last written to. This parameter can be NULL if the application does not want to set 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

Not all file systems can record creation and last access time and not all file systems record them in the same manner. For example, on Windows NT FAT, create time has a resolution of 10 milliseconds, write time has a resolution of 2

seconds, and access time has a resolution of 1 day (really, the access date). On NTFS, access time has a resolution of 1 hour. Therefore, the **GetFileTime** function may not return the same file time information set using **SetFileTime**. Furthermore, FAT records times on disk in local time. However, NTFS records times on disk in UTC, so it is not affected by changes in time zone or daylight saving time.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, FILETIME, GetFileSize, GetFileTime, GetFileType

---

1.358 SetHandleCount

The **SetHandleCount** function sets the number of file handles available to a process.

Windows NT/2000 and Windows 95/98: This function has no effect, because there is no explicit file handle limit for applications on these platforms.

Win32s: There are only 20 file handles available to a process by default; however you could use **SetHandleCount** to allow a process to use up to 255 file handles.

```
SetHandleCount: procedure
(
    uNumber:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetHandleCount@4" );
```

Parameters

uNumber

[in] Specifies the number of file handles needed by the application.

Return Values

Windows NT/2000 and Windows 95/98: This function simply returns the value specified in the *uNumber* parameter.

Win32s: The return value specifies the number of file handles actually available to the application. It may be fewer than the number specified by the *uNumber* parameter.

Remarks

Windows NT/2000 and Windows 95/98: The maximum number of files that an application can open is determined by the amount of available non-paged memory pool, because each open file handle requires non-paged memory.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions

1.359 SetHandleInformation

The **SetHandleInformation** function sets certain properties of an object handle. The information is specified as a set of bit flags.

```
SetHandleInformation: procedure
(
    hObject:    dword;
    dwMask:     dword;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetHandleInformation@12" );
```

Parameters

hObject

[in] Handle to an object. The **SetHandleInformation** function sets information associated with this object handle.

You can specify a handle to one of the following types of objects: access token, event, file, file mapping, job, mailslot, mutex, pipe, printer, process, registry key, semaphore, serial communication device, socket, thread, or waitable timer.

Windows 2000: This parameter can also be a handle to a console input buffer or a console screen buffer.

dwMask

[in] A mask that specifies the bit flags to be changed. Use the same flag constants shown in the description of *dwFlags*.

dwFlags

[in] A set of bit flags that specify properties of the object handle. This parameter can be one of the following values.

Value	Meaning
HANDLE_FLAG_INHERIT	If this flag is set, a child process created with the *bInheritHandles* parameter of **CreateProcess** set to TRUE will inherit the object handle.
HANDLE_FLAG_PROTECT_FROM_CLOSE	If this flag is set, calling the **CloseHandle** function will not close the object handle.

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 or clear the associated bit flag in *dwFlags*, you must set a change mask bit flag in *dwMask*.

Requirements

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

Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Handles and Objects Overview, Handle and Object Functions, CreateProcess, CloseHandle, GetHandleInformation

---

1.360 SetInformationJobObject

The **SetInformationJobObject** function sets limits for a job object.

```
SetInformationJobObject: procedure
(
        hJob:                   dword;
        JobObjectInfoClass:     JOBOBJECTINFOCLASS;
    var lpJobObjectInfo:        var;
        cbJobObjectInfoLength:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetInformationJobObject@16" );
```

Parameters

hJob

[in] Handle to the job whose limits are being set. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_SET_ATTRIBUTES access right associated with it. For more information, see Job Object Security and Access Rights.

JobObjectInfoClass

[in] Specifies the information class for limits to be set. This parameter can be one of the following values.

Value	Meaning
JobObjectAssociateCompletionPortInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_ASSOCIATE_COMPLETION_PORT** structure.
JobObjectBasicLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JobObjectBasicUIRestrictions	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_UI_RESTRICTIONS** structure.
JobObjectEndOfJobTimeInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_END_OF_JOB_TIME_INFORMATION** structure.
JobObjectExtendedLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure.
JobObjectSecurityLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_SECURITY_LIMIT_INFORMATION** structure. The *hJob* handle must have the JOB_OBJECT_SET_SECURITY_ATTRIBUTES access right associated with it.

lpJobObjectInfo

[in] Specifies the limits to be set for the job. The format of this data depends on the value of *JobObjectInfoClass*.

cbJobObjectInfoLength

[in] Specifies the count, in bytes, of the job information being set.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

You can use the **SetInformationJobObject** function to set several limits in a single call. If you want to establish the limits one at a time or change a subset of the limits, call the **QueryInformationJobObject** function to obtain the current limits, modify these limits, and then call **SetInformationJobObject**.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, JOBOBJECT_ASSOCIATE_COMPLETION_PORT, JOBOBJECT_BASIC_LIMIT_INFORMATION, JOBOBJECT_BASIC_UI_RESTRICTIONS, JOBOBJECT_END_OF_JOB_TIME_INFORMATION, JOBOBJECT_EXTENDED_LIMIT_INFORMATION, JOBOBJECT_SECURITY_LIMIT_INFORMATION , QueryInformationJobObject

---

1.361 SetLastError

The **SetLastError** function sets the last-error code for the calling thread.

```
SetLastError: procedure
(
    dwErrCode:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetLastError@4" );
```

Parameters

dwErrCode

[in] Specifies the last-error code for the thread.

Return Values

This function does not return a value.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by your application and to ensure that your error code does not conflict with any sys-

tem-defined error codes.

This function is intended primarily for dynamic-link libraries (DLL). Calling this function after an error occurs lets the DLL emulate the behavior of the Win32 API.

Most Win32 functions call **SetLastError** when they fail. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference topic.

Applications can retrieve the value saved by this function by using the **GetLastError** function. The use of **GetLastError** is optional; an application can call it to find out the specific reason for a function failure.

The last-error code is kept in thread local storage so that multiple threads do not overwrite each other's values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions, GetLastError, SetLastErrorEx

---

1.362 SetLocalTime

The **SetLocalTime** function sets the current local time and date.

```
SetLocalTime: procedure
(
    var lpSystemTime:   SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetLocalTime@4" );
```

Parameters

lpSystemTime

[in] Pointer to a **SYSTEMTIME** structure that contains the current local date and time.

The **wDayOfWeek** member of the **SYSTEMTIME** structure 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 NT/2000: The system uses UTC internally. Therefore, when you call **SetLocalTime**, Windows NT/Windows 2000 uses the current time zone information, including the daylight saving time setting, to perform the conversion. Note that Windows NT/Windows 2000 uses the daylight saving time setting of the current time, not the new time you are setting. Therefore, calling **SetLocalTime** again, now that the daylight saving time setting is set for the new time, will guarantee the correct result.

Windows NT/2000: The **SetLocalTime** function enables the SE_SYSTEMTIME_NAME privilege before changing the local time. This privilege is disabled by default. For more information about security privileges, see Privileges.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, AdjustTokenPrivileges, GetLocalTime, GetSystemTime, SetSystemTimeAdjustment, SYSTEMTIME

1.363 SetLocaleInfo

The **SetLocaleInfo** function sets an item of locale information. This setting only affects the user override portion of the locale settings; it does not set the system defaults.

```
SetLocaleInfo: procedure
(
    Locale:     LCID;
    LCType:     LCTYPE;
    lpLCData:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetLocaleInfoA@12" );
```

Parameters

Locale

[in] Specifies the locale whose information the function will set. The locale provides a context for the string mapping or sort key generation. An application can use the **MAKELCID** macro to create a locale identifier.

LCType

[in] Specifies the type of locale information to be set by the function. Note that only one **LCTYPE** may be specified per call. Not all **LCTYPE** values are valid; see the list of valid **LCTYPE** values in the following Remarks section.

lpLCData

[in] Pointer to a null-terminated string containing the locale information the function will set. The information must be in the specified **LCTYPE**'s particular format.

Return Values

If the function succeeds, the return values is a nonzero value.

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_ACCESS

ERROR_INVALID_FLAGS

ERROR_INVALID_PARAMETER

Remarks

The locale information is always passed in as a null-terminated Unicode string in the Unicode version of the function, and as a null-terminated ANSI string in the ANSI version. No integers are allowed by this function; any numeric values must be specified as Unicode or ANSI text. Each LCTYPE has a particular format, as noted in **Locales**. Note that

several of the LCTYPE values are linked together, so that changing one changes another value as well. For more information, see Locale Information.

Only the following **LCTYPE** values are valid for this function:

LOCALE_ICALENDARTYPE	LOCALE_SDATE
LOCALE_ICURRDIGITS	LOCALE_SDECIMAL
LOCALE_ICURRENCY	LOCALE_SGROUPING
LOCALE_IDIGITS	LOCALE_SLIST
LOCALE_IFIRSTDAYOFWEEK	LOCALE_SLONGDATE
LOCALE_IFIRSTWEEKOFYEAR	LOCALE_SMONDECIMALSEP
LOCALE_ILZERO	LOCALE_SMONGROUPING
LOCALE_IMEASURE	LOCALE_SMONTHOUSANDSEP
LOCALE_INEGCURR	LOCALE_SNEGATIVESIGN
LOCALE_INEGNUMBER	LOCALE_SPOSITIVESIGN
LOCALE_IPAPERSIZE	LOCALE_SSHORTDATE
LOCALE_ITIME	LOCALE_STHOUSAND
LOCALE_S1159	LOCALE_STIME
LOCALE_S2359	LOCALE_STIMEFORMAT
LOCALE_SCURRENCY	LOCALE_SYEARMONTH

Windows 2000: The ANSI version of this function will fail if it is used with a Unicode-only locale. See Language Identifiers.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, GetLocaleInfo, MAKELCID

1.364 SetMailslotInfo

The **SetMailslotInfo** function sets the time-out value used by the specified mailslot for a read operation.

```
SetMailslotInfo: procedure
(
    hMailslot:      dword;
    lReadTimeout:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetMailslotInfo@8" );
```

Parameters

hMailslot

[in] Handle to a mailslot. The **CreateMailslot** function must create this handle.

lReadTimeout

[in] Specifies the amount of time, in milliseconds, a read operation can wait for a message to be written to the mailslot before a time-out occurs. The following values have special meanings.

Value	Meaning
0	Returns immediately if no message is present. (The system does not treat an immediate return as an error.)
MAILSLOT_WAIT_FOREVER	Waits forever for a message.

This time-out value applies to all subsequent read operations and to all inherited mailslot handles.

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 initial time-out value used by a mailslot for a read operation is typically set by **CreateMailslot** when the mailslot is created.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Mailslots Overview, Mailslot Functions, CreateMailslot, GetMailslotInfo

---

1.365 SetNamedPipeHandleState

The **SetNamedPipeHandleState** function sets the read mode and the blocking mode of the specified named pipe. If the specified handle is to the client end of a named pipe and if the named pipe server process is on a remote computer, the function can also be used to control local buffering.

```
SetNamedPipeHandleState: procedure
(
        hNamedPipe:             dword;
    var lpMode:                 dword;
    var lpMaxCollectionCount:   dword;
    var lpCollectDataTimeout:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetNamedPipeHandleState@16" );
```

Parameters

hNamedPipe

[in] Handle to the named pipe instance. This parameter can be a handle to the server end of the pipe, as returned by the **CreateNamedPipe** function, or to the client end of the pipe, as returned by the **CreateFile** function. The handle must have GENERIC_WRITE access to the named pipe.

Windows NT/2000: This parameter can also be a handle to an anonymous pipe, as returned by the **CreatePipe** function.

lpMode

[in] Pointer to a variable that supplies the new mode. The mode is a combination of a read-mode flag and a wait-mode flag. This parameter can be NULL if the mode is not being set. Specify one of the following modes.

Mode	Description
PIPE_READMODE_BYTE	Data is read from the pipe as a stream of bytes. This mode is the default if no read-mode flag is specified.
PIPE_READMODE_MESSAGE	Data is read from the pipe as a stream of messages. The function fails if this flag is specified for a byte-type pipe.

One of the following wait modes can be specified:

Mode	Description
PIPE_WAIT	Blocking mode is enabled. This mode is the default if no wait-mode flag is specified. When a blocking mode pipe handle is specified in the **`ReadFile, WriteFile,`** or **`ConnectNamedPipe`** function, operations are not finished until there is data to read, all data is written, or a client is connected. Use of this mode can mean waiting indefinitely in some situations for a client process to perform an action.
PIPE_NOWAIT	Nonblocking mode is enabled. In this mode, **ReadFile**, **WriteFile**, and **ConnectNamedPipe** always return immediately. Note that nonblocking mode is supported for compatibility with Microsoft® LAN Manager version 2.0 and should not be used to achieve asynchronous input and output (I/O) with named pipes.

lpMaxCollectionCount

[in] Pointer to a variable that specifies the maximum number of bytes collected on the client computer before transmission to the server. This parameter must be NULL if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same machine. This parameter is ignored if the client process specifies the FILE_FLAG_WRITE_THROUGH flag in the **CreateFile** function when the handle was created. This parameter can be NULL if the collection count is not being set.

lpCollectDataTimeout

[in] Pointer to a variable that specifies the maximum time, in milliseconds, that can pass before a remote named pipe transfers information over the network. This parameter must be NULL if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same computer. This parameter is ignored if the client process specified the FILE_FLAG_WRITE_THROUGH flag in the **`CreateFile`** function when the handle was created. This parameter can be NULL if the collection count is not being set.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, ConnectNamedPipe, CreateFile, CreateNamedPipe, GetNamedPipeHandleState, ReadFile, WriteFile

1.366 SetPriorityClass

The **SetPriorityClass** function sets the priority class for the specified process. This value together with the priority value of each thread of the process determines each thread's base priority level.

```
SetPriorityClass: procedure
(
    hProcess:           dword;
    dwPriorityClass:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetPriorityClass@8" );
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have the PROCESS_SET_INFORMATION access right. For more information, see Process Security and Access Rights.

dwPriorityClass

[in] Specifies the priority class for the process. This parameter can be one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Specify this class for a process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS	Specify this class for a process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Specify this class for a process with no special scheduling needs.

REALTIME_PRIORITY_CLASS	Specify this class for a process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

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

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. The **SetThreadPriority** function enables setting the base priority level of a thread relative to the priority class of its process. For more information, see Scheduling Priorities.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, CreateThread, GetPriorityClass, GetThreadPriority, SetThreadPriority

---

1.367 SetProcessAffinityMask

The **SetProcessAffinityMask** function sets a processor affinity mask for the threads of the specified process.

A process affinity mask is a bit vector in which each bit represents the processor on which the threads of the process are allowed to run.

The value of the process affinity mask must be a proper subset of the mask values obtained by the **GetProcessAffinityMask** function.

```
SetProcessAffinityMask: procedure
(
    hProcess:               dword;
    dwProcessAffinityMask:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetProcessAffinityMask@8" );
```

Parameters

hProcess

[in] Handle to the process whose affinity mask is to be set. This handle must have the PROCESS_SET_INFORMATION access right. For more information, see Process Security and Access Rights.

dwProcessAffinityMask

[in] Specifies an affinity mask for the threads of the process.

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

Process affinity is inherited by any process that you start with the **CreateProcess** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, GetProcessAffinityMask

---

1.368 SetProcessPriorityBoost

The **SetProcessPriorityBoost** function disables the ability of the system to temporarily boost the priority of the threads of the specified process.

```
SetProcessPriorityBoost: procedure
(
    hProcess:               dword;
    DisablePriorityBoost:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetProcessPriorityBoost@8" );
```

Parameters

hProcess

[in] Handle to the process. This handle must have the PROCESS_SET_INFORMATION access right. For more information, see Process Security and Access Rights.

DisablePriorityBoost

[in] Specifies the priority boost control state. A value of TRUE indicates that dynamic boosting is to be disabled. A value of FALSE restores normal behavior.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

When a thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetProcessPriorityBoost** is called with the *DisablePriorityBoost* parameter set to TRUE, its threads' priorities are not boosted. This setting affects all existing threads and any threads subsequently created by the process. To restore normal behavior, call **SetProcessPriorityBoost** with *DisablePriorityBoost* set to FALSE.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetProcessPriorityBoost

---

1.369 SetProcessShutdownParameters

The **SetProcessShutdownParameters** function sets shutdown parameters for the currently calling process. This function sets a shutdown order for a process relative to the other processes in the system.

```
SetProcessShutdownParameters: procedure
(
    dwLevel:    dword;
    dwFlags:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetProcessShutdownParameters@8" );
```

Parameters

dwLevel

[in] Specifies the shutdown priority for a process relative to other processes in the system. The system shuts down processes from high *dwLevel* values to low. The highest and lowest shutdown priorities are reserved for system components. This parameter must be in the following range of values.

Value	Meaning
000–0FF	System reserved last shutdown range.
100–1FF	Application reserved last shutdown range.
200–2FF	Application reserved "in between" shutdown range.
300–3FF	Application reserved first shutdown range.
400–4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

dwFlags

[in] This parameter can be the following value.

Value	Meaning
SHUTDOWN_NORETRY	Specifies whether to retry the shutdown if the specified time-out period expires. If this flag is specified, the system terminates the process without displaying a retry dialog box for the user.

Return Values

If the function is succeeds, the return value is nonzero.

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

Remarks

Applications running in the system security context do not get shut down by the operating system. They get notified of shutdown or logoff through the callback function installable via **SetConsoleCtrlHandler**. They also get notified in the order specified by the *dwLevel* parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetProcessShutdownParameters, SetConsoleCtrl-Handler

---

1.370 SetProcessWorkingSetSize

The **SetProcessWorkingSetSize** function sets the minimum and maximum working set sizes for the specified process.

The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The size of the working set of a process is specified in bytes. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

```
SetProcessWorkingSetSize: procedure
(
    hProcess:                   dword;
    dwMinimumWorkingSetSize:    SIZE_T;
    dwMaximumWorkingSetSize:    SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetProcessWorkingSetSize@12" );
```

Parameters

hProcess

[in] Handle to the process whose working set sizes is to be set.

Windows NT/2000: The handle must have PROCESS_SET_QUOTA access rights. For more information, see Process Security and Access Rights.

dwMinimumWorkingSetSize

[in] Specifies a minimum working set size for the process. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value -1, the function temporarily trims the working set of the specified process to zero. This essentially swaps the process out of physical RAM memory.

dwMaximumWorkingSetSize

[in] Specifies a maximum working set size for the process. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active and memory is in short supply.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value -1, the function temporarily trims the working set of the specified process to zero. This essentially swaps the process out of physical RAM memory.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. Call **GetLastError** to obtain extended error information.

Remarks

The working set of the specified process can be emptied by specifying the value -1 for both the minimum and maximum working set sizes.

If the values of either *dwMinimumWorkingSetSize* or *dwMaximumWorkingSetSize* are greater than the process' current working set sizes, the specified process must have the SE_INC_BASE_PRIORITY_NAME privilege. Users in the Administrators and Power Users groups generally have this privilege. For more information about security privileges, see Privileges.

The operating system allocates working set sizes on a first-come, first-served basis. For example, if an application successfully sets 40 megabytes as its minimum working set size on a 64-megabyte system, and a second application requests a 40-megabyte working set size, the operating system denies the second application's request.

Using the **SetProcessWorkingSetSize** function to set an application's minimum and maximum working set sizes does not guarantee that the requested memory will be reserved, or that it will remain resident at all times. When the application is idle, or a low-memory situation causes a demand for memory, the operating system can reduce the application's working set. An application can use the **VirtualLock** function to lock ranges of the application's virtual address space in memory; however, that can potentially degrade the performance of the system.

When you increase the working set size of an application, you are taking away physical memory from the rest of the system. This can degrade the performance of other applications and the system as a whole. It can also lead to failures of operations that require physical memory to be present; for example, creating processes, threads, and kernel pool. Thus, you must use the **SetProcessWorkingSetSize** function carefully. You must always consider the performance of the whole system when you are designing an application.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetProcessWorkingSetSize, VirtualLock

---

1.371 SetStdHandle

The **SetStdHandle** function sets the handle for the standard input, standard output, or standard error device.

```
SetStdHandle: procedure
(
    nStdHandle: dword;
    hHandle:    dword
);
    stdcall;
    returns( "eax" );
```

```
    external( "__imp__SetStdHandle@8" );
```

Parameters

nStdHandle

[in] Specifies the standard handle to be set. This parameter can be one of the following values.

Value	Meaning
STD_INPUT_HANDLE	Standard input handle
STD_OUTPUT_HANDLE	Standard output handle
STD_ERROR_HANDLE	Standard error handle

hHandle

[in] Handle to set as standard input, standard output, or standard error.

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 standard handles of a process may have been redirected by a call to **SetStdHandle**, in which case **GetStdHandle** will return the redirected handle. If the standard handles have been redirected, you can specify the CONIN$ value in a call to the **CreateFile** function to get a handle to a console's input buffer. Similarly, you can specify the CONOUT$ value to get a handle to the console's active screen buffer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CreateFile, GetStdHandle

---

1.372 SetSystemPowerState

The **SetSystemPowerState** function suspends the system by shutting power down. Depending on the *ForceFlag* parameter, the function either suspends operation immediately or requests permission from all applications and device drivers before doing so.

The calling process must have the SE_SHUTDOWN_NAME privilege. To enable the SE_SHUTDOWN_NAME privilege, use the **AdjustTokenPrivileges** function. For more information, see Privileges.

```
SetSystemPowerState: procedure
(
    fSuspend:   boolean;
    fForce:     boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetSystemPowerState@8" );
```

Parameters

fSuspend

Windows NT/2000: [in] Specifies the state of the system. If TRUE, the system is suspended. If FALSE, the system hibernates.

Windows 95/98: Ignored.

fForce

[in] Forced suspension. If TRUE, the function broadcasts a **PBT_APMSUSPEND** event to each application and driver, then immediately suspends operation. If FALSE, the function broadcasts a **PBT_APMQUERYSUSPEND** event to each application to request permission to suspend operation.

Return Values

If power has been suspended and subsequently restored, the return value is nonzero.

If the system was not suspended, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If any application or driver denies permission to suspend operation, the function broadcasts a **PBT_APMQUERYSUSPENDFAILED** event to each application and driver. If power is suspended, this function returns only after system operation is resumes and related **WM_POWERBROADCAST** messages have been broadcast to all applications and drivers.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Power Management Overview, Power Management Functions, PBT_APMQUERYSUSPEND, PBT_APMQUERYSUSPENDFAILED, PBT_APMSUSPEND, WM_POWERBROADCAST

1.373 SetSystemTime

The **SetSystemTime** function sets the current system time and date. The system time is expressed in Coordinated Universal Time (UTC).

```
SetSystemTime: procedure
(
    var lpSystemTime:   SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetSystemTime@4" );
```

Parameters

lpSystemTime

[in] Pointer to a **SYSTEMTIME** structure that contains the current system date and time.

The **wDayOfWeek** member of the **SYSTEMTIME** structure 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 NT/2000: The **SetSystemTime** function enables the SE_SYSTEMTIME_NAME privilege before changing the system time. This privilege is disabled by default. For more information about security privileges, see Privileges.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetSystemTime, SetSystemTimeAdjustment, SYSTEMTIME

1.374 SetSystemTimeAdjustment

The **SetSystemTimeAdjustment** function enables or disables periodic time adjustments to the system's time-of-day clock. Such time adjustments are used to synchronize the time of day with some other source of time information. When periodic time adjustments are enabled, they are applied at each clock interrupt.

```
SetSystemTimeAdjustment: procedure
(
    dwTimeAdjustment:           dword;
    bTimeAdjustmentDisabled:    boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetSystemTimeAdjustment@8" );
```

Parameters

dwTimeAdjustment

[in] Specifies the number of 100-nanosecond units added to the time-of-day clock at each clock interrupt if periodic time adjustment is enabled.

bTimeAdjustmentDisabled

[in] Specifies the time adjustment mode that the system is to use. Periodic system time adjustments can be disabled or enabled.

A value of TRUE specifies that periodic time adjustment is to be disabled. The system is free to adjust time of day using its own internal mechanisms. The value of *dwTimeAdjustment* is ignored. The system's internal adjustment mechanisms may cause the time-of-day clock to jump noticeably when adjustments are made.

A value of FALSE specifies that periodic time adjustment is to be enabled, and will be used to adjust the time-of-day clock. The system will not interfere with the time adjustment scheme, and will not attempt to synchronize time of day on its own. The system will add the value of *dwTimeAdjustment* to the time of day at each clock interrupt.

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**. One way the function can fail is if the caller does not possess the SE_SYSTEMTIME_NAME privilege.

Remarks

The **GetSystemTimeAdjustment** and **SetSystemTimeAdjustment** functions support algorithms that synchronize the time-of-day clock, reported via **GetSystemTime** and **GetLocalTime**, with another time source using a periodic time adjustment.

The **SetSystemTimeAdjustment** function supports two modes of time synchronization: time-adjustment – disabled and time-adjustment – enabled.

In the first mode, *bTimeAdjustmentDisabled* is set to FALSE. At each clock interrupt, the system adds the value of *dwTimeAdjustment* to the time of day. The clock interrupt rate may be determined by calling **GetSystemTimeAdjustment**, and looking at the returned value of the **DWORD** value pointed to by *lpTimeIncrement*.

In the second mode, *bTimeAdjustmentDisabled* is set to TRUE. At each clock interrupt, the system adds the interval between clock interrupts to the time of day. No adjustment to that interval is made. The system is free to periodically refresh the time-of-day clock using other techniques. Such other techniques may cause the time-of-day clock to jump noticeably when adjustments are made.

An application must have system-time privilege (the SE_SYSTEMTIME_NAME privilege) for this function to succeed. The SE_SYSTEMTIME_NAME privilege is disabled by default. Use the **AdjustTokenPrivileges** function to enable the privilege before calling **SetSystemTimeAdjustment**, and then to disable the privilege after the **SetSystemTimeAdjustment** call.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, AdjustTokenPrivileges, GetLocalTime, GetSystemTime, GetSystemTimeAdjustment

---

1.375 SetTapeParameters

The **SetTapeParameters** function either specifies the block size of a tape or configures the tape device.

```
SetTapeParameters: procedure
(
        hDevice:            dword;
        dwOperation:        dword;
    var lpTapeInformation:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetTapeParameters@12" );
```

Parameters

hDevice

[in] Handle to the device for which to set configuration information. This handle is created by using the **CreateFile** function.

dwOperation

[in] Specifies the type of information to set. This parameter must be one of the following values.

Value	Description
SET_TAPE_MEDIA_INFORMATION	Sets the tape-specific information specified by the *lpTapeInformation* parameter.
SET_TAPE_DRIVE_INFORMATION	Sets the device-specific information specified by *lpTapeInformation*.

lpTapeInformation

[in] Pointer to a structure that contains the information to set. If the *dwOperation* parameter is SET_TAPE_MEDIA_INFORMATION, *lpTapeInformation* points to a **TAPE_SET_MEDIA_PARAMETERS** structure.

If *dwOperation* is SET_TAPE_DRIVE_INFORMATION, *lpTapeInformation* points to a **TAPE_SET_DRIVE_PARAMETERS** structure.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Requirements

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

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, GetTapeParameters, TAPE_SET_DRIVE_PARAMETERS, TAPE_SET_MEDIA_PARAMETERS

1.376 SetTapePosition

The **SetTapePosition** sets the tape position on the specified device.

```
SetTapePosition: procedure
(
    hDevice:            dword;
    dwPositionMethod:   dword;
    dwPartition:        dword;
    dwOffsetLow:        dword;
    dwOffsetHigh:       dword;
    bImmediate:         boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetTapePosition@24" );
```

Parameters

hDevice

[in] Handle to the device on which to set the tape position. This handle is created by using the **`CreateFile`** function.

dwPositionMethod

[in] Specifies the type of positioning to perform. This parameter must be one of the following values.

Value	Meaning
TAPE_ABSOLUTE_BLOCK	Moves the tape to the device-specific block address specified by the *dwOffsetLow* and *dwOffsetHigh* parameters. The *dwPartition* parameter is ignored.
TAPE_LOGICAL_BLOCK	Moves the tape to the block address specified by *dwOffsetLow* and *dwOffsetHigh* in the partition specified by *dwPartition*.
TAPE_REWIND	Moves the tape to the beginning of the current partition. The *dwPartition*, *dwOffsetLow*, and *dwOffsetHigh* parameters are ignored.
TAPE_SPACE_END_OF_DATA	Moves the tape to the end of the data on the partition specified by *dwPartition*.
TAPE_SPACE_FILEMARKS	Moves the tape forward (or backward) the number of filemarks specified by *dwOffsetLow* and *dwOffsetHigh* in the current partition. The *dwPartition* parameter is ignored.

TAPE_SPACE_RELATIVE_BLOCKS	Moves the tape forward (or backward) the number of blocks specified by *dwOffsetLow* and *dwOffsetHigh* in the current partition. The *dwPartition* parameter is ignored.
TAPE_SPACE_SEQUENTIAL_FMKS	Moves the tape forward (or backward) to the first occurrence of *n* filemarks in the current partition, where *n* is the number specified by *dwOffsetLow* and *dwOffsetHigh*. The *dwPartition* parameter is ignored.
TAPE_SPACE_SEQUENTIAL_SMKS	Moves the tape forward (or backward) to the first occurrence of *n* setmarks in the current partition, where *n* is the number specified by *dwOffsetLow* and *dwOffsetHigh*. The *dwPartition* parameter is ignored.
TAPE_SPACE_SETMARKS	Moves the tape forward (or backward) the number of setmarks specified by *dwOffsetLow* and *dwOffsetHigh* in the current partition. The *dwPartition* parameter is ignored.

dwPartition

[in] Specifies the partition to position within. If *dwPartition* is zero, the current partition is used. Partitions are numbered logically from 1 through *n*, where 1 is the first partition on the tape and *n* is the last.

dwOffsetLow

[in] Specifies the low-order bits of the block address or count for the position operation specified by the *dwPositionMethod* parameter.

dwOffsetHigh

[in] Specifies the high-order bits of the block address or count for the position operation specified by the *dwPositionMethod* parameter. If the high-order bits are not required, this parameter should be zero.

bImmediate

[in] Indicates whether to return as soon as the move operation begins. If this parameter is TRUE, the function returns immediately; if FALSE, the function does not return until the move operation has been completed.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.

ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

If the offset specified by *dwOffsetLow* and *dwOffsetHigh* specifies the number of blocks, filemarks, or setmarks to move, a positive offset moves the tape forward to the end of the last block, filemark, or setmark. A negative offset moves the tape backward to the beginning of the last block, filemark, or setmark. If the offset is zero, the tape does not move.

To obtain information about the status, capabilities, and capacities of tape drives and media, call the **GetTapeParameters** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile, GetTapeParameters, GetTapePosition

---

1.377 SetThreadAffinityMask

The **SetThreadAffinityMask** function sets a processor affinity mask for the specified thread.

A thread affinity mask is a bit vector in which each bit represents the processors that a thread is allowed to run on.

A thread affinity mask must be a proper subset of the process affinity mask for the containing process of a thread. A thread is only allowed to run on the processors its process is allowed to run on.

```
SetThreadAffinityMask: procedure
(
    hThread:                dword;
    dwThreadAffinityMask:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadAffinityMask@8" );
```

Parameters

hThread

[in] Handle to the thread whose affinity mask is to be set.

Windows NT/2000: This handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see Thread Security and Access Rights.

dwThreadAffinityMask

Windows NT/2000: [in] Specifies an affinity mask for the thread.

Windows 95/98: [in] This value must be 1.

Return Values

If the function succeeds, the return value is nonzero.

Windows NT/2000: The return value is the thread's previous affinity mask.

Windows 95/98: The return value is 1. To succeed, *hThread* must be valid and *dwThreadAffinityMask* must be 1.

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

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetProcessAffinityMask, OpenThread, SetThread-IdealProcessor

---

1.378 SetThreadContext

The **SetThreadContext** function sets the context for the specified thread.

```
SetThreadContext: procedure
(
        hThread:    dword;
    var lpContext:  CONTEXT
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadContext@8" );
```

Parameters

hThread

[in] Handle to the thread whose context is to be set.

Windows NT/ 2000: The handle must have the THREAD_SET_CONTEXT access right to the thread. For more information, see Thread Security and Access Rights.

lpContext

[in] Pointer to the **CONTEXT** structure that contains the context to be set in the specified thread. The value of the **ContextFlags** member of this structure specifies which portions of a thread's context to set. Some values in the **CONTEXT** structure that cannot be specified are silently set to the correct value. This includes bits in the CPU status register that specify the privileged processor mode, global enabling bits in the debugging register, and other states that must be controlled by the operating system.

Return Values

If the context was set, the return value is nonzero.

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

Remarks

The function allows the selective context to be set based on the value of the **ContextFlags** member of the context structure. The thread handle identified by the *hThread* parameter is typically being debugged, but the function can also operate even when it is not being debugged.

Do not try to set the context for a running thread; the results are unpredictable. Use the **SuspendThread** function to suspend the thread before calling **SetThreadContext**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, CONTEXT, GetThreadContext, SuspendThread

---

1.379 SetThreadExecutionState

The **SetThreadExecutionState** function enables applications to inform the system that it is in use, thereby preventing the system from entering the sleeping power state while the application is running.

```
SetThreadExecutionState: procedure
(
    esFlags:    EXECUTION_STATE
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadExecutionState@4" );
```

Parameters

esFlags

[in] Specifies the thread's execution requirements. This parameter can be one or more of the following values.

Flag	Description
ES_SYSTEM_REQUIRED	Informs the system that the thread is performing some operation that is not normally detected as activity by the system.
ES_DISPLAY_REQUIRED	Informs the system that the thread is performing some operation that is not normally detected as display activity by the system.
ES_USER_PRESENT	Informs the system that a user is present. If a user is present, the system will use the power management policies set by the user. Otherwise, the system will not wake the display device and will return to the sleeping state as soon as possible.
ES_CONTINUOUS	Informs the system that the state being set should remain in effect until the next call that uses ES_CONTINUOUS and one of the other state flags is cleared.

Return Values

If the function succeeds, the return value is the previous thread execution state.

If the function fails, the return value is NULL.

Remarks

Activities that are automatically detected include local keyboard or mouse input, server activity, and changing window focus. Activities that are not automatically detected include disk or CPU activity and video display.

Calling **SetThreadExecutionState** with ES_SYSTEM_REQUIRED prevents the system from putting the computer in the sleeping state by resetting the system idle timer. Calling **SetThreadExecutionState** with ES_DISPLAY_REQUIRED prevents the system from turning off the display by resetting the display idle timer. Calling **SetThreadExecutionState** without ES_CONTINUOUS simply resets the idle timer; to keep the display or system in the working state, the thread must call **SetThreadExecutionState** periodically.

To run properly on a power-managed computer, applications such as fax servers, answering machines, backup agents, and network management applications must use ES_SYSTEM_REQUIRED | ES_CONTINUOUS when they process events. Multimedia applications, such as video players and presentation applications, must use ES_DISPLAY_REQUIRED when they display video for long periods of time without user input. Applications such as word processors, spreadsheets, browsers, and games do not need to call **SetThreadExecutionState**.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Power Management Overview, Power Management Functions, SetSystemPowerState, WM_POWERBROADCAST

1.380 SetThreadIdealProcessor

The **SetThreadIdealProcessor** function sets a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.

```
SetThreadIdealProcessor: procedure
(
    hThread:            dword;
    dwIdealProcessor:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadIdealProcessor@8" );
```

Parameters

hThread

[in] Handle to the thread whose preferred processor is to be set. The handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see Thread Security and Access Rights.

dwIdealProcessor

[in] Specifies the number of the preferred processor for the thread. A value of MAXIMUM_PROCESSORS tells the system that the thread has no preferred processor.

Return Values

If the function succeeds, the return value is the previous preferred processor or MAXIMUM_PROCESSORS if the

thread does not have a preferred processor.

If the function fails, the return value is — 1. To get extended error information, call **GetLastError**.

Remarks

You can use the **GetSystemInfo** function to determine the number of processors on the computer. You can also use the **GetProcessAffinityMask** function to check the processors on which the thread is allowed to run. Note that **GetProcessAffinityMask** returns a bit mask whereas **SetThreadIdealProcessor** uses an integer value to represent the processor.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetProcessAffinityMask, GetSystemInfo, OpenThread, SetThreadAffinityMask

---

1.381 SetThreadLocale

The **SetThreadLocale** function sets the calling thread's current locale.

```
SetThreadLocale: procedure
(
    Locale: LCID
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadLocale@4" );
```

Parameters

Locale

[in] Specifies the new locale for the calling thread. This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values.

Value	Meaning
LOCALE_SYSTEM_DEFAULT	Default system locale.
LOCALE_USER_DEFAULT	Default user locale.

For more information, see **Locales**.

Return Values

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

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

Remarks

When a thread is created, it uses the system default thread locale. The system reads the system default thread locale from the registry when the system boots. This system default can be modified for future process and thread creation using Control Panel's International application.

The **SetThreadLocale** function affects the selection of resources that are defined with a **LANGUAGE** statement. This affects such functions as **`CreateDialog, DialogBox, LoadMenu, LoadString,`** and **`FindResource`**, and sets the code page implied by CP_THREAD_ACP, but does not affect **`FindResourceEx`**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

National Language Support Overview, National Language Support Functions, CreateDialog, DialogBox, LoadMenu, LoadString, FindResource, GetThreadLocale, GetSystemDefaultLCID, GetUserDefaultLCID, MAKELCID

1.382 SetThreadPriority

The **SetThreadPriority** function sets the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base priority level.

```
SetThreadPriority: procedure
(
    hThread:    dword;
    nPriority:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadPriority@8" );
```

Parameters

hThread

[in] Handle to the thread whose priority value is to be set.

Windows NT/2000: The handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see Thread Security and Access Rights.

nPriority

[in] Specifies the priority value for the thread. This parameter can be one of the following values:

Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.

THREAD_PRIORITY_IDLE	Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.
THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for REALTIME_PRIORITY_CLASS processes.

Windows 2000: This parameter can also be -7, -6, -5, -4, -3, 3, 4, 5, or 6. For more information, see Scheduling Priorities.

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

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level does scheduling of threads at a lower level take place.

The **SetThreadPriority** function enables setting the base priority level of a thread relative to the priority class of its process. For example, specifying THREAD_PRIORITY_HIGHEST in a call to **SetThreadPriority** for a thread of an IDLE_PRIORITY_CLASS process sets the thread's base priority level to 6. For a table that shows the base priority levels for each combination of priority class and thread priority value, see Scheduling Priorities.

For IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, and HIGH_PRIORITY_CLASS processes, the system dynamically boosts a thread's base priority level when events occur that are important to the thread. REALTIME_PRIORITY_CLASS processes do not receive dynamic boosts.

All threads initially start at THREAD_PRIORITY_NORMAL. Use the **GetPriorityClass** and **SetPriorityClass** functions to get and set the priority class of a process. Use the **GetThreadPriority** function to get the priority value of a thread.

Use the priority class of a process to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. Use thread priority values to differentiate the relative priorities of the tasks of a process. For example, a thread that handles input for a window could have a higher priority level than a thread that performs intensive calculations for the CPU.

When manipulating priorities, be very careful to ensure that a high-priority thread does not consume all of the available CPU time. A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using REALTIME_PRIORITY_CLASS may cause disk caches to not flush, hang the mouse, and so on.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetPriorityClass, GetThreadPriority, SetPriority-Class

---

1.383 SetThreadPriorityBoost

The **SetThreadPriorityBoost** function disables the ability of the system to temporarily boost the priority of a thread.

```
SetThreadPriorityBoost: procedure
(
    hThread:                dword;
    DisablePriorityBoost:   boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetThreadPriorityBoost@8" );
```

Parameters

hThread

[in] Handle to the thread whose priority is to be boosted. This thread must have the THREAD_SET_INFORMATION access right associated with it. For more information, see Thread Security and Access Rights.

DisablePriorityBoost

[in] Specifies the priority boost control state. A value of TRUE indicates that dynamic boosting is to be disabled. A value of FALSE restores normal behavior.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

When a thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetThreadPriorityBoost** is called with the *DisablePriorityBoost* parameter set to TRUE, the thread's priority is not boosted. To restore normal behavior, call **SetThreadPriorityBoost** with *DisablePriorityBoost* set to FALSE.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, OpenThread, GetThreadPriorityBoost

1.384 SetTimeZoneInformation

The **SetTimeZoneInformation** function sets the current time-zone parameters. These parameters control translations from Coordinated Universal Time (UTC) to local time.

```
SetTimeZoneInformation: procedure
(
    lpTimeZoneInformation:  TIME_ZONE_INFORMATION
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetTimeZoneInformation@4" );
```

Parameters

lpTimeZoneInformation

[in] Pointer to a **TIME_ZONE_INFORMATION** structure that contains the time-zone parameters to set.

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

All translations between UTC and local time are based on the following formula:

```
UTC = local time + bias
```

The bias is the difference, in minutes, between UTC and local time.

To disable the automatic adjustment for daylight saving time, use the following registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\TimeZoneInformation\DisableAutoDaylightTimeSet

Remarks

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetTimeZoneInformation, TIME_ZONE_INFORMATION

1.385 SetUnhandledExceptionFilter

The **SetUnhandledExceptionFilter** function enables an application to supersede the top-level exception handler of each thread and process.

After calling this function, if an exception occurs in a process that is not being debugged, and the exception makes it to the unhandled exception filter, that filter will call the exception filter function specified by the *lpTopLevelExceptionFilter* parameter.

```
SetUnhandledExceptionFilter: procedure
```

```
(
    lpTopLevelExceptionFilter:  TOP_LEVEL_EXCEPTION_FILTER
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetUnhandledExceptionFilter@4" );
```

Parameters

lpTopLevelExceptionFilter

[in] Pointer to a top-level exception filter function that will be called whenever the **UnhandledExceptionFilter** function gets control, and the process is not being debugged. A value of NULL for this parameter specifies default handling within **UnhandledExceptionFilter**.

The filter function has syntax congruent to that of **UnhandledExceptionFilter**: It takes a single parameter of type **LPEXCEPTION_POINTERS**, and returns a value of type **LONG**. The filter function should return one of the following values.

Value	Meaning
EXCEPTION_EXECUTE_HANDLER	Return from **UnhandledExceptionFilter** and execute the associated exception handler. This usually results in process termination.
EXCEPTION_CONTINUE_EXECUTION	Return from **UnhandledExceptionFilter** and continue execution from the point of the exception. Note that the filter function is free to modify the continuation state by modifying the exception information supplied through its **LPEXCEPTION_POINTERS** parameter.
EXCEPTION_CONTINUE_SEARCH	Proceed with normal execution of **UnhandledExceptionFilter**. That means obeying the **SetErrorMode** flags, or invoking the Application Error pop-up message box.

Return Values

The **SetUnhandledExceptionFilter** function returns the address of the previous exception filter established with the function. A NULL return value means that there is no current top-level exception handler.

Remarks

Issuing **SetUnhandledExceptionFilter** replaces the existing top-level exception filter for all existing and all future threads in the calling process.

The exception handler specified by *lpTopLevelExceptionFilter* is executed in the context of the thread that caused the fault. This can affect the exception handler's ability to recover from certain exceptions, such as an invalid stack.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Structured Exception Handling Overview, Structured Exception Handling Functions, UnhandledExceptionFilter

---

1.386 SetVolumeLabel

The **SetVolumeLabel** function sets the label of a file system volume.

```
SetVolumeLabel: procedure
(
    lpRootPathName: string;
    lpVolumeName:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetVolumeLabelA@8" );
```

Parameters

lpRootPathName

[in] Pointer to a null-terminated string specifying the root directory of a file system volume. This is the volume the function will label. A trailing backslash is required. If this parameter is NULL, the root of the current directory is used.

lpVolumeName

[in] Pointer to a string specifying a name for the volume. If this parameter is NULL, the function deletes the label from the specified volume.

Return Values

If the function succeeds, the return value is nonzero.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, GetVolumeInformation

1.387 SetWaitableTimer

The **SetWaitableTimer** function activates the specified waitable timer. When the due time arrives, the timer is signaled and the thread that set the timer calls the optional completion routine.

```
SetWaitableTimer: procedure
(
        hTimer:                 dword;
    var pDueTime:               LARGE_INTEGER;
        lPeriod:                dword;
        pfnCompletionRoutine:   procedure;
    var lpArgToCompletionRoutine: var;
        fResume:                boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetWaitableTimer@24" );
```

Parameters

hTimer

[in] Handle to the timer object. The **CreateWaitableTimer** or **OpenWaitableTimer** function returns this handle.

pDueTime

[in] Specifies when the state of the timer is to be set to signaled, in 100 nanosecond intervals. Use the format described by the **FILETIME** structure. Positive values indicate absolute time. Be sure to use a UTC-based absolute time, as the system uses UTC-based time internally. Negative values indicate relative time. The actual timer accuracy depends on the capability of your hardware. For more information about UTC-based time, see System Time.

lPeriod

[in] Specifies the period of the timer, in milliseconds. If *lPeriod* is zero, the timer is signaled once. If *lPeriod* is greater than zero, the timer is periodic. A periodic timer automatically reactivates each time the period elapses, until the timer is canceled using the **CancelWaitableTimer** function or reset using **SetWaitableTimer**. If *lPeriod* is less than zero, the function fails.

pfnCompletionRoutine

[in] Pointer to an optional completion routine. The completion routine is application-defined function of type PTIMERAPCROUTINE to be executed when the timer is signaled. For more information on the timer callback function, see **TimerAPCProc**.

lpArgToCompletionRoutine

[in] Pointer to the structure that is passed to the optional completion routine.

fResume

[in] Specifies whether to restore a system in suspended power conservation mode when the timer state is set to signaled. If *fResume* is TRUE on a platform that does not support a restore, the call will succeed, but **GetLastError** returns ERROR_NOT_SUPPORTED.

Return Value

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

Timers are initially inactive. To activate a timer, call **SetWaitableTimer**. If the timer is already active when you call **SetWaitableTimer**, the timer is stopped, then it is reactivated. Stopping the timer in this manner does not set the timer state to signaled, so threads blocked in a wait operation on the timer remain blocked.

When the specified due time arrives, the timer becomes inactive and the APC is queued to the thread that set the timer. The state of the timer is set to signaled, the timer is reactivated using the specified period, and the thread that set the timer calls the completion routine when it enters an alertable wait state. For more information, see **QueueUserAPC**.

If the thread that set the timer exits before the timer elapses, the timer is cancelled. If you call **SetWaitableTimer** on a timer that has been set by another thread and that thread is not in an alertable state, the completion routine is cancelled.

When a manual-reset timer is set to the signaled state, it remains in this state until **SetWaitableTimer** is called to reset the timer. As a result, a periodic manual-reset timer is set to the signaled state when the initial due time arrives and remains signaled until it is reset. When a synchronization timer is set to the signaled state, it remains in this state until a thread completes a wait operation on the timer object.

Example

For an example that uses **SetWaitableTimer**, see Using Waitable Timer Objects.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98 or later.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, CancelWaitableTimer, CreateWaitableTimer, FILETIME, OpenWaitableTimer, TimerAPCProc

---

1.388 SetupComm

The **SetupComm** function initializes the communications parameters for a specified communications device.

```
SetupComm: procedure
(
    hFile:      dword;
    dwInQueue:  dword;
    dwOutQueue: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SetupComm@12" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

dwInQueue

[in] Specifies the recommended size, in bytes, of the device's internal input buffer.

dwOutQueue

[in] Specifies the recommended size, in bytes, of the device's internal output buffer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

After a process uses the **CreateFile** function to open a handle to a communications device, it can call **SetupComm** to set the communications parameters for the device. If it does not set them, the device uses the default parameters when the first call to another communications function occurs.

The *dwInQueue* and *dwOutQueue* parameters specify the recommended sizes for the internal buffers used by the driver for the specified device. For example, YMODEM protocol packets are slightly larger than 1024 bytes. Therefore, a recommended buffer size might be 1200 bytes for YMODEM communications. For Ethernet-based communications, a recommended buffer size might be 1600 bytes, which is slightly larger than a single Ethernet frame.

The device driver receives the recommended buffer sizes, but is free to use any input and output (I/O) buffering scheme, as long as it provides reasonable performance and data is not lost due to overrun (except under extreme circumstances). For example, the function can succeed even though the driver does not allocate a buffer, as long as some other portion of the system provides equivalent functionality.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, SetCommState

---

1.389 SignalObjectAndWait

The **SignalObjectAndWait** function allows the caller to atomically signal an object and wait on another object.

```
SignalObjectAndWait: procedure
(
    hObjectToSignal:    dword;
    hObjectToWaitOn:    dword;
    dwMilliseconds:     dword;
    bAlertable:         boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SignalObjectAndWait@16" );
```

Parameters

hObjectToSignal

[in] Specifies the handle to the object to signal. This object can be a semaphore, a mutex, or an event.

Windows NT/2000: If the handle is a semaphore, SEMAPHORE_MODIFY_STATE access is required. If the handle is an event, EVENT_MODIFY_STATE access is required. If the handle is a mutex, SYNCHRONIZE access is assumed, because only the owner of a mutex may release it. For more information, see Synchronization Object Security and Access Rights.

hObjectToWaitOn

[in] Specifies the handle to the object to wait for. For a list of the object types whose handles you can specify, see the Remarks section.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled and no completion or asynchronous procedure call (APC) objects are queued. If *dwMilliseconds* is zero, the function tests the object's state, checks for queued completion routines or APCs, and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

bAlertable

[in] Specifies whether the function returns when the system queues an I/O completion routine or an APC for the calling thread. If TRUE, the function returns and the thread calls the completion routine or APC function. If FALSE, the function does not return, and the thread does not call the completion routine or APC function.

A completion routine is queued when the **ReadFileEx** or **WriteFileEx** function in which it was specified has completed. The wait function returns and the completion routine is called only if *bAlertable* is TRUE, and the calling thread is the thread that initiated the read or write operation. An APC is queued when you call **QueueUserAPC**.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following:

Value	Meaning
WAIT_ABANDONED	The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to nonsignaled.
WAIT_IO_COMPLETION	One or more I/O completion routines or user-mode APCs are queued for execution.
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is nonsignaled.

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

Remarks

A completion routine is queued for execution when the **`ReadFileEx`** or **`WriteFileEx`** function in which it was specified has been completed. The wait function returns and the completion routine is executed only if *bAlertable* is TRUE, and the calling thread is the thread that initiated the read or write operation.

The **SignalObjectAndWait** function can wait for the following objects:

Change notification

Console input

Event

Job

Mutex

Process

Semaphore

Thread

Waitable timer

For more information, see Synchronization Objects.

Use caution when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **`MsgWaitForMultipleObjects`** or **`MsgWaitForMultipleObjectsEx`**, rather than **SignalObjectAndWait**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx

---

1.390 SizeofResource

The **SizeofResource** function returns the size, in bytes, of the specified resource.

```
SizeofResource: procedure
(
    hModule:    dword;
    hResInfo:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SizeofResource@8" );
```

Parameters

hModule

[in] Handle to the module whose executable file contains the resource.

hResInfo

[in] Handle to the resource. This handle must be created by using the **FindResource** or **FindResourceEx** function.

Return Values

If the function succeeds, the return value is the number of bytes in the resource.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, FindResource, FindResourceEx

---

1.391 Sleep

The **Sleep** function suspends the execution of the current thread for the specified interval.

To enter an alertable wait state, use the **SleepEx** function.

```
Sleep: procedure
(
    dwMilliseconds: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__Sleep@4" );
```

Parameters

dwMilliseconds

[in] Specifies the time, in milliseconds, for which to suspend execution. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. A value of INFINITE causes an infinite delay.

Return Values

This function does not return a value.

Remarks

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **Sleep** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **Sleep** with infinite delay, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **`MsgWaitForMultipleObjects`** or **`MsgWaitForMultipleObjectsEx`**, rather than **Sleep**.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, SleepEx

---

1.392 SleepEx

The **SleepEx** function suspends the current thread until one of the following occurs:

An I/O completion callback function is called

An asynchronous procedure call (APC) is queued to the thread.

The time-out interval elapses

```
SleepEx: procedure
(
    dwMilliseconds: dword;
    bAlertable:     boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__SleepEx@8" );
```

Parameters

dwMilliseconds

[in] Specifies the time, in milliseconds, that the delay is to occur. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. A value of INFINITE causes an infinite delay.

bAlertable

[in] Specifies whether the function may terminate early due to an I/O completion callback function or an APC. If *bAlertable* is FALSE, the function does not return until the time-out period has elapsed. If an I/O completion callback occurs, the function does not return and the I/O completion function is not executed. If an APC is queued to the thread, the function does not return and the APC function is not executed.

If *bAlertable* is TRUE and the thread that called this function is the same thread that called the extended I/O function (**ReadFileEx** or **WriteFileEx**), the function returns when either the time-out period has elapsed or when an I/O completion callback function occurs. If an I/O completion callback occurs, the I/O completion function is called. If an APC is queued to the thread (**QueueUserAPC**), the function returns when either the timer-out period has elapsed or when the APC function is called.

Return Values

The return value is zero if the specified time interval expired.

The return value is WAIT_IO_COMPLETION if the function returned due to one or more I/O completion callback functions. This can happen only if *bAlertable* is TRUE, and if the thread that called the **SleepEx** function is the same thread that called the extended I/O function.

Remarks

This function can be used with the **ReadFileEx** or **WriteFileEx** functions to suspend a thread until an I/O operation has been completed. These functions specify a completion routine that is to be executed when the I/O operation has been completed. For the completion routine to be executed, the thread that called the I/O function must be in an alertable wait state when the completion callback function occurs. A thread goes into an alertable wait state by calling either **SleepEx**, **MsgWaitForMultipleObjectsEx, WaitForSingleObjectEx,** or **WaitForMultipleObjectsEx**, with the function's *bAlertable* parameter set to TRUE.

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **SleepEx** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **SleepEx** with infinite delay, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than **SleepEx**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, QueueUserAPC, ReadFileEx, Sleep, WaitForMultipleObjectsEx, WaitForSingleObjectEx, WriteFileEx

---

1.393 SuspendThread

The **SuspendThread** function suspends the specified thread.

```
SuspendThread: procedure
(
    hThread:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SuspendThread@4" );
```

Parameters

hThread

[in] Handle to the thread that is to be suspended.

Windows NT/2000: The handle must have THREAD_SUSPEND_RESUME access. For more information, see Thread Security and Access Rights.

Return Values

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is -1. To get extended error information, use the **GetLastError** function.

Remarks

If the function succeeds, execution of the specified thread is suspended and the thread's suspend count is incremented.

Suspending a thread causes the thread to stop executing user-mode (application) code.

Each thread has a suspend count (with a maximum value of MAXIMUM_SUSPEND_COUNT). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling **SuspendThread** causes the target thread's suspend count to be incremented. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

The **ResumeThread** function decrements the suspend count of a suspended thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, OpenThread, ResumeThread

---

1.394 SwitchToFiber

The **SwitchToFiber** function schedules a fiber. The caller of must be a fiber.

```
SwitchToFiber: procedure
(
    lpFiber:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__SwitchToFiber@4" );
```

Parameters

lpFiber

[in] Specifies the address of the fiber to schedule.

Return Values

This function does not return a value.

Remarks

You create fibers with **CreateFiber**. Before you can schedule fibers associated with a thread, you must call **ConvertThreadToFiber** to set up an area in which to save the fiber state information. The thread is now the currently executing fiber.

The **SwitchToFiber** function saves the state information of the current fiber and restores the state of the specified fiber. You can call **SwitchToFiber** with the address of a fiber created by a different thread. To do this, you must have the address returned to the other thread when it called **CreateFiber** and you must use proper synchronization.

Warning Avoid making the following call:

```
SwitchToFiber( GetCurrentFiber() );
```

This call may cause unpredictable problems.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateFiber, ConvertThreadToFiber

---

1.395 SwitchToThread

The **SwitchToThread** function causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

```
SwitchToThread: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__SwitchToThread@0" );
```

Parameters

This function has no parameters.

Return Values

If calling the **SwitchToThread** function causes the operating system to switch execution to another thread, the return value is nonzero.

If there are no other threads ready to execute, the operating system does not switch execution to another thread, and the return value is zero.

Remarks

The yield of execution is in effect for up to one thread-scheduling time slice. After that, the operating system reschedules execution for the yielding thread. The rescheduling is determined by the priority of the yielding thread and the status of other threads that are available to run.

Note The yield of execution is limited to the processor of the calling thread. The operating system will not switch execution to another processor, even if that processor is idle or is running a thread of lower priority.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, SuspendThread

1.396 SystemTimeToFileTime

The **SystemTimeToFileTime** function converts a system time to a file time.

```
SystemTimeToFileTime: procedure
(
    var lpSystemTime:   SYSTEMTIME;
    var lpFileTime:     FILETIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__SystemTimeToFileTime@8" );
```

Parameters

lpSystemTime

[in] Pointer to a **SYSTEMTIME** structure that contains the time to be converted.

The **wDayOfWeek** member of the **SYSTEMTIME** structure is ignored.

lpFileTime

[out] Pointer to a **FILETIME** structure to receive the converted system time.

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

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, DosDateTimeToFileTime, FILETIME, FileTimeToDosDateTime, FileTimeToSystemTime, SYSTEMTIME

1.397 SystemTimeToTzSpecificLocalTime

The **SystemTimeToTzSpecificLocalTime** function converts a time in Coordinated Universal Time (UTC) to a specified time zone's corresponding local time.

```
SystemTimeToTzSpecificLocalTime: procedure
(
    var lpTimeZone:         TIME_ZONE_INFORMATION;
    var lpUniversalTime:    SYSTEMTIME;
    var lpLocalTime:        SYSTEMTIME
);
    stdcall;
    returns( "eax" );
    external( "__imp__SystemTimeToTzSpecificLocalTime@12" );
```

Parameters

lpTimeZone

[in] Pointer to a **TIME_ZONE_INFORMATION** structure that specifies the time zone of interest.

If *lpTimeZone* is NULL, the function uses the currently active time zone.

lpUniversalTime

[in] Pointer to a **SYSTEMTIME** structure that specifies a UTC. The function converts this universal time to the specified time zone's corresponding local time.

lpLocalTime

[out] Pointer to a **SYSTEMTIME** structure that receives the local time information.

Return Values

If the function succeeds, the return value is nonzero, and the function sets the members of the **SYSTEMTIME** structure pointed to by *lpLocalTime* to the appropriate local time values.

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

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Header: Declared in kenel32.hhf
Library: Use Kernel32.lib.

See Also

Time Overview, Time Functions, GetSystemTime, GetTimeZoneInformation, SYSTEMTIME, TIME_ZONE_INFORMATION

1.398 TerminateJobObject

The **TerminateJobObject** function terminates all processes currently associated with the job.

```
TerminateJobObject: procedure
(
    hJob:       dword;
    uExitCode:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__TerminateJobObject@8" );
```

Parameters

hJob

[in] Handle to the job whose processes will be terminated. The **CreateJobObject** or **OpenJobObject** function returns this handle. This handle must have the JOB_OBJECT_TERMINATE access right. For more information, see Job Object Security and Access Rights.

uExitCode

[in] Specifies the exit code for the processes and threads terminated as a result of this call.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

It is not possible for any of the processes associated with the job to postpone or handle the termination. It is as if **TerminateProcess** were called for each process associated with the job.

Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateJobObject, OpenJobObject, TerminateProcess

---

1.399 TerminateProcess

The **TerminateProcess** function terminates the specified process and all of its threads.

```
TerminateProcess: procedure
(
    hProcess:   dword;
    uExitCode:  dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__TerminateProcess@8" );
```

Parameters

hProcess

[in] Handle to the process to terminate.

Windows NT/2000: The handle must have PROCESS_TERMINATE access. For more information, see Process Security and Access Rights.

uExitCode

[in] Specifies the exit code for the process and for all threads terminated as a result of this call. Use the **GetExitCodeProcess** function to retrieve the process's exit value. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

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 **TerminateProcess** function is used to unconditionally cause a process to exit. Use it only in extreme circumstances. The state of global data maintained by dynamic-link libraries (DLLs) may be compromised if **TerminateProcess** is used rather than **ExitProcess**.

TerminateProcess causes all threads within a process to terminate, and causes a process to exit, but DLLs attached to

the process are not notified that the process is terminating.

Terminating a process causes the following:

All of the object handles opened by the process are closed.

All of the threads in the process terminate their execution.

The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.

The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.

The termination status of the process changes from STILL_ACTIVE to the exit value of the process.

Terminating a process does not cause child processes to be terminated.

Terminating a process does not necessarily remove the process object from the system. A process object is deleted when the last handle to the process is closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, ExitProcess, OpenProcess, GetExitCodeProcess, GetExitCodeThread

---

1.400 TerminateThread

The **TerminateThread** function terminates a thread.

```
TerminateThread: procedure
(
    hThread:    dword;
    dwExitCode: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__TerminateThread@8" );
```

Parameters

hThread

[in/out] Handle to the thread to terminate.

Windows NT/2000: The handle must have THREAD_TERMINATE access. For more information, see Thread Security and Access Rights.

dwExitCode

[in] Specifies the exit code for the thread. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

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

TerminateThread is used to cause a thread to exit. When this occurs, the target thread has no chance to execute any user-mode code and its initial stack is not deallocated. DLLs attached to the thread are not notified that the thread is terminating.

TerminateThread is a dangerous function that should only be used in the most extreme cases. You should call **TerminateThread** only if you know exactly what the target thread is doing, and you control all of the code that the target thread could possibly be running at the time of the termination. For example, **TerminateThread** can result in the following problems:

If the target thread owns a critical section, the critical section will not be released.

If the target thread is executing certain kernel32 calls when it is terminated, the kernel32 state for the thread's process could be inconsistent.

If the target thread is manipulating the global state of a shared DLL, the state of the DLL could be destroyed, affecting other users of the DLL.

A thread cannot protect itself against **TerminateThread**, other than by controlling access to its handles. The thread handle returned by the **CreateThread** and **CreateProcess** functions has THREAD_TERMINATE access, so any caller holding one of these handles can terminate your thread.

If the target thread is the last thread of a process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from STILL_ACTIVE to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the system. A thread object is deleted when the last thread handle is closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, CreateThread, ExitThread, GetExitCodeThread, OpenThread

---

1.401 Thread32First

Retrieves information about the first thread of any process encountered in a system snapshot.

```
Thread32First: procedure
(
        hSnapshot:  dword;
    var lpte:       THREADENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Thread32First@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpte

[in/out] Pointer to a **THREADENTRY32** structure.

Return Values

Returns TRUE if the first entry of the thread list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no threads exist or the snapshot does not contain thread information.

Remarks

The calling application must set the **dwSize** member of **THREADENTRY32** to the size, in bytes, of the structure. **Thread32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other threads recorded in the same snapshot, use the **Thread32Next** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions, , CreateToolhelp32Snapshot, THREADENTRY32, Thread32Next

---

1.402 Thread32Next

Retrieves information about the next thread of any process encountered in the system memory snapshot.

```
Thread32Next: procedure
(
        hSnapshot:  dword;
    var lpte:       THREADENTRY32
);
    stdcall;
    returns( "eax" );
    external( "__imp__Thread32Next@8" );
```

Parameters

hSnapshot

[in] Handle to the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpte

[out] Pointer to a **THREADENTRY32** structure.

Return Values

Returns TRUE if the next entry of the thread list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no threads exist or the snapshot does not contain thread information.

Remarks

To retrieve information about the first thread recorded in a snapshot, use the **Thread32First** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.403 TlsAlloc

The **TlsAlloc** function allocates a thread local storage (TLS) index. Any thread of the process can subsequently use this index to store and retrieve values that are local to the thread.

```
TlsAlloc: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__TlsAlloc@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a TLS index initialized to zero.

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

Remarks

The threads of the process can use the TLS index in subsequent calls to the **`TlsFree, TlsSetValue,`** or **`TlsGetValue`** functions.

TLS indexes are typically allocated during process or dynamic-link library (DLL) initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot. To store a value in its slot, a thread specifies the index in a call to **TlsSetValue**. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems.

Windows 2000: There is a limit of 1088 TLS indexes per process. **Windows NT 4.0 and earlier:** There is a limit of 64 TLS indexes per process.

TLS indexes are not valid across process boundaries. A DLL cannot assume that an index assigned in one process is valid in another process.

A DLL might use **TlsAlloc**, **`TlsSetValue, TlsGetValue,`** and **`TlsFree`** as follows:

When a DLL attaches to a process, the DLL uses **TlsAlloc** to allocate a TLS index. The DLL then allocates some dynamic storage and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the initial thread of the process. The TLS index is stored in a global or static variable of the DLL.

Each time the DLL attaches to a new thread of the process, the DLL allocates some dynamic storage for the new thread and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the new thread.

Each time an initialized thread makes a DLL call requiring the data in its dynamic storage, the DLL uses the TLS index in a call to **TlsGetValue** to retrieve the address of the dynamic storage for that thread.

For additional information on thread local storage, see Thread Local Storage.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, TlsFree, TlsGetValue, TlsSetValue

---

1.404 TlsFree

The **TlsFree** function releases a thread local storage (TLS) index, making it available for reuse.

```
TlsFree: procedure
(
    dwTlsIndex: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__TlsFree@4" );
```

Parameters

dwTlsIndex

[in] Specifies a TLS index that was allocated by the **TlsAlloc** function.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the threads of the process have allocated dynamic storage and used the TLS index to store pointers to this storage, they should free the storage before calling **TlsFree**. The **TlsFree** function does not free any dynamic storage that has been associated with the TLS index. It is expected that DLLs call this function (if at all) only during their process detach routine.

For a brief discussion of typical uses of the TLS functions, see the Remarks section of the **TlsAlloc** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, TlsAlloc, TlsGetValue, TlsSetValue

1.405 TlsGetValue

The **TlsGetValue** function retrieves the value in the calling thread's thread local storage (TLS) slot for the specified TLS index. Each thread of a process has its own slot for each TLS index.

```
TlsGetValue: procedure
(
    dwTlsIndex: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__TlsGetValue@4" );
```

Parameters

dwTlsIndex

[in] Specifies a TLS index that was allocated by the **TlsAlloc** function.

Return Values

If the function succeeds, the return value is the value stored in the calling thread's TLS slot associated with the specified index.

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

Note The data stored in a TLS slot can have a value of zero. In this case, the return value is zero and **GetLastError** returns NO_ERROR.

Remarks

TLS indexes are typically allocated by the **TlsAlloc** function during process or DLL initialization. After it is allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this function succeeds if *dwTlsIndex* is in the range 0 through (TLS_MINIMUM_AVAILABLE – 1). It is up to the programmer to ensure that the index is valid.

Functions that return indications of failure call **SetLastError** when they fail. They generally do not call **SetLastError** when they succeed. The **TlsGetValue** function is an exception to this general rule. The **TlsGetValue** function calls **SetLastError** to clear a thread's last error when it succeeds. That allows checking for the error-free retrieval of NULL values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, GetLastError, SetLastError, TlsAlloc, TlsFree, TlsSetValue

1.406 TlsSetValue

The **TlsSetValue** function stores a value in the calling thread's thread local storage (TLS) slot for the specified TLS index. Each thread of a process has its own slot for each TLS index.

```
TlsSetValue: procedure
(
        dwTlsIndex: dword;
    var lpTlsValue: var
);
    stdcall;
    returns( "eax" );
    external( "__imp__TlsSetValue@8" );
```

Parameters

dwTlsIndex

[in] Specifies a TLS index that was allocated by the **TlsAlloc** function.

lpTlsValue

[in] Specifies the value to be stored in the calling thread's TLS slot specified by *dwTlsIndex*.

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

TLS indexes are typically allocated by the **TlsAlloc** function during process or DLL initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this function succeeds if *dwTlsIndex* is in the range 0 through (TLS_MINIMUM_AVAILABLE – 1). It is up to the programmer to ensure that the index is valid.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, TlsAlloc, TlsFree, TlsGetValue

1.407 Toolhelp32ReadProcessMemory

Copies memory allocated to another process into an application-supplied buffer.

```
Toolhelp32ReadProcessMemory: procedure
(
        th32ProcessID:          dword;
    var lpBaseAddress:          var;
    var lpBuffer:               var;
```

```
        cbRead:                 dword;
    var lpNumberOfBytesRead:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__Toolhelp32ReadProcessMemory@20" );
```

Parameters

th32ProcessID

[in] Identifier of the process whose memory is being copied. This parameter can be zero to copy the memory of the current process.

lpBaseAddress

[in] Base address in the specified process to read. Before transferring any data, the system verifies that all data in the base address and memory of the specified size is accessible for read access. If this is the case, the function proceeds. Otherwise, the function fails.

lpBuffer

[out] Pointer to a buffer that receives the contents of the address space of the specified process.

cbRead

[in] Number of bytes to read from the specified process.

lpNumberOfBytesRead

[out] Number of bytes copied to the specified buffer. If this parameter is NULL, it is ignored.

Return Values

Returns TRUE if successful.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tool Help Library Overview, Tool Help Functions

1.408 TransactNamedPipe

The **TransactNamedPipe** function combines the functions that write a message to and read a message from the specified named pipe into a single network operation.

```
TransactNamedPipe: procedure
(
        hNamedPipe:     dword;
    var lpInBuffer:     var;
        nInBufferSize:  dword;
    var lpOutBuffer:    var;
        nOutBufferSize: dword;
    var lpBytesRead:    dword;
    var lpOverlapped:   OVERLAPPED
);
```

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

Parameters

hNamedPipe

[in] Handle to the named pipe returned by the **CreateNamedPipe** or **CreateFile** function.

Windows NT/2000: This parameter can also be a handle to an anonymous pipe, as returned by the **CreatePipe** function.

lpInBuffer

[in] Pointer to the buffer containing the data written to the pipe.

nInBufferSize

[in] Specifies the size, in bytes, of the write buffer.

lpOutBuffer

[out] Pointer to the buffer that receives the data read from the pipe.

nOutBufferSize

[in] Specifies the size, in bytes, of the read buffer.

lpBytesRead

[out] Pointer to the variable that receives the number of bytes read from the pipe.

If *lpOverlapped* is NULL, *lpBytesRead* cannot be NULL.

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

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure. This structure is required if *hNamedPipe* was opened with FILE_FLAG_OVERLAPPED.

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

If *hNamePipe* was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, **TransactNamedPipe** is executed as an overlapped operation. The **OVERLAPPED** structure should contain a manual-reset event object (which can be created by using the **CreateEvent** function). If the operation cannot be completed immediately, **TransactNamedPipe** returns FALSE and **GetLastError** returns ERROR_IO_PENDING. In this situation, the event object is set to the nonsignaled state before **TransactNamedPipe** returns, and it is set to the signaled state when the transaction has finished. For more information about overlapped operations, see **Pipes**.

If *hNamedPipe* was not opened with FILE_FLAG_OVERLAPPED, **TransactNamedPipe** does not return until the operation is complete.

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

TransactNamedPipe fails if the server did not create the pipe as a message-type pipe or if the pipe handle is not in

message-read mode. For example, if a client is running on the same machine as the server and uses the \\.\pipe*pipename* format to open the pipe, the pipe is opened in byte mode by the named pipe file system (NPFS). If the client uses the form *server*\pipe*pipename*, the redirector opens the pipe in message mode. A byte mode pipe handle can be changed to message-read mode with the **SetNamedPipeHandleState** function.

The function cannot be completed successfully until data is written into the buffer specified by the *lpOutBuffer* parameter. The *lpOverlapped* parameter is available to enable the calling thread to perform other tasks while the operation is executing in the background.

If the message to be read is longer than the buffer specified by the *nOutBufferSize* parameter, **TransactNamedPipe** returns FALSE and the **GetLastError** function returns ERROR_MORE_DATA. The remainder of the message can be read by a subsequent call to **ReadFile**, **ReadFileEx**, or **PeekNamedPipe**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CreateEvent, CreateFile, CreateNamedPipe, GetOverlappedResult, GetQueuedCompletionStatus, PeekNamedPipe, ReadFile, ReadFileEx, SetNamedPipeHandleState, OVERLAPPED

1.409 TransmitCommChar

The **TransmitCommChar** function transmits a specified character ahead of any pending data in the output buffer of the specified communications device.

```
TransmitCommChar: procedure
(
    hFile:  dword;
    cChar:  char
);
    stdcall;
    returns( "eax" );
    external( "__imp__TransmitCommChar@8" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

cChar

[in] Specifies the character to be transmitted.

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 **TransmitCommChar** function is useful for sending an interrupt character (such as a CTRL+C) to a host system.

If the device is not transmitting, **TransmitCommChar** cannot be called repeatedly. Once **TransmitCommChar** places a character in the output buffer, the character must be transmitted before the function can be called again. If the previous character has not yet been sent, **TransmitCommChar** returns an error.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, WaitCommEvent

---

1.410 TryEnterCriticalSection

The **TryEnterCriticalSection** function attempts to enter a critical section without blocking. If the call is successful, the calling thread takes ownership of the critical section.

```
TryEnterCriticalSection: procedure
(
    var lpCriticalSection:  CRITICAL_SECTION
);
    stdcall;
    returns( "eax" );
    external( "__imp__TryEnterCriticalSection@4" );
```

Parameters

lpCriticalSection

[in/out] Specifies the critical section object.

Return Values

If the critical section is successfully entered or the current thread already owns the critical section, the return value is nonzero.

If another thread already owns the critical section, the return value is zero.

Remarks

The threads of a single process can use a critical section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type **CRITICAL_SECTION**. Before using a critical section, some thread of the process must call the **`InitializeCriticalSection`** or **`InitializeCriticalSectionAndSpinCount`** function to initialize the object.

To enable mutually exclusive use of a shared resource, each thread calls the **`EnterCriticalSection`** or **TryEnterCriticalSection** function to request ownership of the critical section before executing any section of code that uses the protected resource. The difference is that **TryEnterCriticalSection** returns immediately, regardless of whether it obtained ownership of the critical section, while **EnterCriticalSection** blocks until the thread can take ownership of the critical section. When it has finished executing the protected code, the thread uses the **`LeaveCriticalSection`** function to relinquish ownership, enabling another thread to become the owner and gain access to the protected resource. The thread must call **LeaveCriticalSection** once for each time that it entered the critical section.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.

Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, DeleteCriticalSection, EnterCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection

---

1.411 UnhandledExceptionFilter

The **UnhandledExceptionFilter** function passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an **Application Error** message box and causes the exception handler to be executed. This function can be called only from within the filter expression of an exception handler.

```
UnhandledExceptionFilter: procedure
(
    var ExceptionInfo:  _EXCEPTION_POINTERS
);
    stdcall;
    returns( "eax" );
    external( "__imp__UnhandledExceptionFilter@4" );
```

Parameters

ExceptionInfo

[in] Pointer to an **EXCEPTION_POINTERS** structure containing a description of the exception and the processor context at the time of the exception. This pointer is the return value of a call to the **GetExceptionInformation** function.

Return Values

The function returns one of the following values.

Value	Meaning
EXCEPTION_CONTINUE_SEARCH	The process is being debugged, so the exception should be passed (as second chance) to the application's debugger.
EXCEPTION_EXECUTE_HANDLER	If the SEM_NOGPFAULTERRORBOX flag was specified in a previous call to **SetErrorMode**, no Application Error message box is displayed. The function returns control to the exception handler, which is free to take any appropriate action.

Remarks

If the process is not being debugged, the function displays an Application Error message box, depending on the current error mode. The default behavior is to display the dialog box, but this can be disabled by specifying SEM_NOGPFAULTERRORBOX in a call to the **SetErrorMode** function.

The system uses **UnhandledExceptionFilter** internally to handle exceptions that occur during process and thread creation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Structured Exception Handling Overview, Structured Exception Handling Functions, EXCEPTION_POINTERS, GetExceptionInformation, SetErrorMode, SetUnhandledExceptionFilter, UnhandledExceptionFilter

---

1.412 UnlockFile

The **UnlockFile** function unlocks a region in an open file. Unlocking a region enables other processes to access the region.

For an alternate way to specify the region, use the **UnlockFileEx** function.

```
UnlockFile: procedure
(
    hFile:                      dword;
    dwFileOffsetLow:            dword;
    dwFileOffsetHigh:           dword;
    nNumberOfBytesToUnlockLow:  dword;
    nNumberOfBytesToUnlockHigh: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__UnlockFile@20" );
```

Parameters

hFile

[in] Handle to a file that contains a region locked with **LockFile**. The file handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwFileOffsetLow

[in] Specifies the low-order word of the starting byte offset in the file where the locked region begins.

dwFileOffsetHigh

[in] Specifies the high-order word of the starting byte offset in the file where the locked region begins.

Windows 95/98: *dwFileOffsetHigh* must be 0, the sign extension of the value of *dwFileOffsetLow*. Any other value will be rejected.

nNumberOfBytesToUnlockLow

[in] Specifies the low-order word of the length of the byte range to be unlocked.

nNumberOfBytesToUnlockHigh

[in] Specifies the high-order word of the length of the byte range to be unlocked.

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

Unlocking a region of a file releases a previously acquired lock on the file. The region to unlock must correspond exactly to an existing locked region. Two adjacent regions of a file cannot be locked separately and then unlocked using a single region that spans both locked regions.

If a process terminates with a portion of a file locked or closes a file that has outstanding locks, the behavior is not specified.

This function works on a file allocation table (FAT) – based file system only if the operating system is running SHARE.EXE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, LockFile, UnlockFileEx

---

1.413 UnlockFileEx

The **UnlockFileEx** function unlocks a previously locked byte range in an open file.

```
UnlockFileEx: procedure
(
        hFile:                      dword;
        dwReserved:                 dword;
        nNumberOfBytesToUnlockLow:  dword;
        nNumberOfBytesToUnlockHigh: dword;
    var lpOverlapped:               OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__UnlockFileEx@20" );
```

Parameters

hFile

[in] Handle to a file that is to have an existing locked region unlocked. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwReserved

Reserved parameter; must be zero.

nNumberOfBytesToUnlockLow

[in] Specifies the low-order part of the length of the byte range to unlock.

nNumberOfBytesToUnlockHigh

[in] Specifies the high-order part of the length of the byte range to unlock.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure that the function uses with the unlocking request. This structure contains the file offset of the beginning of the unlock range.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Unlocking a region of a file releases a previously acquired lock on the file. The region to unlock must correspond

exactly to an existing locked region. Two adjacent regions of a file cannot be locked separately and then unlocked using a single region that spans both locked regions.

If a process terminates with a portion of a file locked or closes a file that has outstanding locks, the behavior is not specified.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, LockFile, LockFileEx, OVERLAPPED, UnlockFile

---

1.414 UnmapViewOfFile

The **UnmapViewOfFile** function unmaps a mapped view of a file from the calling process's address space.

```
UnmapViewOfFile: procedure
(
    var lpBaseAddress:  var
);
    stdcall;
    returns( "eax" );
    external( "__imp__UnmapViewOfFile@4" );
```

Parameters

lpBaseAddress

[in] Pointer to the base address of the mapped view of a file that is to be unmapped. This value must be identical to the value returned by a previous call to the **MapViewOfFile** or **MapViewOfFileEx** function.

Return Values

If the function succeeds, the return value is nonzero, and all dirty pages within the specified range are written "lazily" to disk.

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

Remarks

Although an application may close the file handle used to create a file-mapping object, the system holds the corresponding file open until the last view of the file is unmapped:

Windows 95: Files for which the last view has not yet been unmapped are held open with the same sharing restrictions as the original file handle.

Windows NT/2000: Files for which the last view has not yet been unmapped are held open with no sharing restrictions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File Mapping Overview, File Mapping Functions, MapViewOfFile, MapViewOfFileEx

---

1.415 UpdateResource

The **UpdateResource** function adds, deletes, or replaces a resource in an executable file.

```
UpdateResource: procedure
(
        hUpdate:    dword;
        lpType:     string;
        lpName:     string;
        wLanguage:  word;
    var lpData:     var;
        cbData:     dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__UpdateResourceA@24" );
```

Parameters

hUpdate

[in] Specifies an update-file handle. This handle is returned by the **BeginUpdateResource** function.

lpType

[in] Pointer to a null-terminated string specifying the resource type to be updated. This parameter can also be an integer value passed to the **MAKEINTRESOURCE** macro, or it can be one of the following predefined resource types.

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource
RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry

RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource

lpName

[in] Pointer to a null-terminated string specifying the name of the resource to be updated. This parameter can also be an integer value passed to the **MAKEINTRESOURCE** macro.

wLanguage

[in] Specifies the language identifier of the resource to be updated. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the **MAKELANGID** macro.

lpData

[in] Pointer to the resource data to be inserted into the executable file. If the resource is one of the predefined types, the data must be valid and properly aligned. Note that this is the raw binary data stored in the executable file, not the data provided by **LoadIcon**, **LoadString**, or other resource-specific load functions. All data containing strings or text must be in Unicode format; *lpData* must not point to ANSI data.

If *lpData* is NULL, the specified resource is deleted from the executable file.

cbData

[in] Specifies the size, in bytes, of the resource data at *lpData*.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

An application can use **UpdateResource** repeatedly to make changes to the resource data. Each call to **UpdateResource** contributes to an internal list of additions, deletions, and replacements but does not actually write the data to the executable file. The application must use the **EndUpdateResource** function to write the accumulated changes to the executable file.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Resources Overview, Resource Functions, BeginUpdateResource, EndUpdateResource, LoadIcon, LoadString, LockResource, MAKEINTRESOURCE, MAKELANGID, SizeofResource

---

1.416 VerLanguageName

The **VerLanguageName** function retrieves a description string for the language associated with a specified binary Microsoft language identifier.

```
VerLanguageName: procedure
(
    wLang:      dword;
    szLang:     string;
    nSize:      dword
```

```
);
    stdcall;
    returns( "eax" );
    external( "__imp__VerLanguageNameA@12" );
```

Parameters

wLang

[in] Specifies the binary Microsoft language identifier. For a complete list of the language identifiers supported by Win32, see Language Identifiers.

For example, the description string associated with the language identifier 0x040A is "Spanish (Traditional Sort)". If the identifier is unknown, the *szLang* parameter points to a default string ("Language Neutral").

szLang

[out] Pointer to the buffer that receives the null-terminated string representing the language specified by the *wLang* parameter.

nSize

[in] Specifies the size of the buffer, in characters, pointed to by *szLang*.

Return Values

If the return value is less than or equal to the buffer size, the return value is the size, in characters, of the string returned in the buffer. This value does not include the terminating null character.

If the return value is greater than the buffer size, the return value is the size of the buffer required to hold the entire string. The string is truncated to the length of the existing buffer.

If an error occurs, the return value is zero. Unknown language identifiers do not produce errors.

Remarks

Windows NT 3.51 and earlier: The version information functions work only with Win32 file images. They do not work with 16-bit Windows file images.

Windows 95/98 and Windows NT 4.0 and later: This works on both 16- and 32-bit file images.

Windows 2000 and later: This works on 16-, 32-, and 64-bit file images.

Typically, an installation program uses this function to translate a language identifier returned by the **VerQueryValue** function. The text string may be used in a dialog box that asks the user how to proceed in the event of a language conflict.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Version Information Overview, Version Information Functions, VerQueryValue

---

1.417 VirtualAlloc

The **VirtualAlloc** function reserves or commits a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, unless MEM_RESET is specified.

To allocate memory in the address space of another process, use the **VirtualAllocEx** function.

```
VirtualAlloc: procedure
(
    var lpAddress:          var;
        dwSize:             SIZE_T;
        flAllocationType:   dword;
        flProtect:          dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualAlloc@16" );
```

Parameters

lpAddress

[in] Specifies the desired starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the next 64-kilobyte boundary. If the memory is already reserved and is being committed, the address is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function. If this parameter is NULL, the system determines where to allocate the region.

dwSize

[in] Specifies the size, in bytes, of the region. If the *lpAddress* parameter is NULL, this value is rounded up to the next page boundary. Otherwise, the allocated pages include all pages containing one or more bytes in the range from *lpAddress* to (*lpAddress*+*dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be included in the allocated region.

flAllocationType

[in] Specifies the type of allocation. This parameter can be any combination of the following values.

Value	Meaning
MEM_COMMIT	Allocates physical storage in memory or in the paging file on disk for the specified region of pages. An attempt to commit an already committed page will not cause the function to fail. This means that a range of committed or decommitted pages can be committed without having to worry about a failure.
MEM_PHYSICAL	Allocate physical memory. This value is solely for use with Address Windowing Extensions (AWE) memory.
MEM_RESERVE	Reserves a range of the process's virtual address space without allocating any physical storage. The reserved range cannot be used by any other allocation operations (the **malloc** function, the **LocalAlloc** function, and so on) until it is released. Reserved pages can be committed in subsequent calls to the **VirtualAlloc** function.

MEM_RESET	**Windows NT/2000:** Specifies that the data in the memory range specified by *lpAddress* and *dwSize* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. Setting this value does not guarantee that the range operated on with MEM_RESET will contain zeroes. If you want the range to contain zeroes, decommit the memory and then recommit it. When you specify MEM_RESET, the **VirtualAlloc** function ignores the value of *fProtect*. However, you must still set *fProtect* to a valid protection value, such as PAGE_NOACCESS. **VirtualAlloc** returns an error if you use MEM_RESET and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.
MEM_TOP_DOWN	**Windows NT/2000:** Allocates memory at the highest possible address.
MEM_WRITE_WATCH	**Windows 98:** Causes the system to keep track of the pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE. The write-tracking feature remains enabled for the memory region until the region is freed. To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the **`GetWriteWatch`** function. To reset the write-tracking state, call **GetWriteWatch** or **`ResetWriteWatch`**.

flProtect

[in] Specifies the type of access protection. If the pages are being committed, you can specify any one of the following value, along with PAGE_GUARD and PAGE_NOCACHE as needed.

Value	Meaning
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.

PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages thus act as a one-shot access alarm. PAGE_GUARD is a page protection modifier. An application uses it with one of the other page protection modifiers, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use PAGE_NOACCESS.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This value is a page protection modifier, and it is only valid when used with one of the page protections other than PAGE_NOACCESS.

Return Values

If the function succeeds, the return value is the base address of the allocated region of pages.

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

Remarks

VirtualAlloc can perform the following operations:

Commit a region of pages reserved by a previous call to the **VirtualAlloc** function.

Reserve a region of free pages.

Reserve and commit a region of free pages.

You can use **VirtualAlloc** to reserve a block of pages and then make additional calls to **VirtualAlloc** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

Each page in the process's virtual address space is in one of the following states.

State	Meaning
Free	The page is not committed or reserved and is not accessible to the process. **VirtualAlloc** can reserve, or simultaneously reserve and commit, a free page.
Reserved	The range of addresses cannot be used by other allocation functions, but the page is not accessible and has no physical storage associated with it. **VirtualAlloc** can commit a reserved page, but it cannot reserve it a second time. The **`VirtualFree`** function can release a reserved page, making it a free page.

Committed	Physical storage is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. **VirtualAlloc** can commit an already committed page. This means that you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. **`VirtualFree`** can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page.

If the *lpAddress* parameter is not NULL, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages are allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

Windows NT/2000: The PAGE_GUARD protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see Creating Guard Pages.

Address Windowing Extensions (AWE): The **VirtualAlloc** function can be used to reserve an AWE region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application.

The MEM_PHYSICAL and MEM_RESERVE values must be set in the *AllocationType* parameter. The MEM_COMMIT value must not be set.

The page protection must be set to PAGE_READWRITE.

The **`VirtualFree`** function can be used on an AWE region of memory – in this case, it will invalidate any physical page mappings in the region when freeing the address space. However, the physical pages themselves are not deleted, and the application can subsequently use them. The application must explicitly call **`FreeUserPhysicalPages`** to free the physical pages. On process termination, all resources are automatically cleaned up.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, Address Windowing Extensions (AWE), AllocateUserPhysicalPages, FreeUserPhysicalPages, GetWriteWatch, HeapAlloc, MapUserPhysicalPages, MapUserPhysicalPagesScatter, ResetWriteWatch, VirtualAllocEx, VirtualFree, VirtualLock, VirtualProtect, VirtualQuery

---

1.418 VirtualAllocEx

The **VirtualAllocEx** function reserves, commits, or reserves and commits a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero, unless MEM_RESET is used.

```
VirtualAllocEx: procedure
(
        hProcess:           dword;
    var lpAddress:          var;
        dwSize:             SIZE_T;
        flAllocationType:   dword;
        flProtect:          dword
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__VirtualAllocEx@20" );
```

Parameters

hProcess

[in] Handle to a process. The function allocates memory within the virtual address space of this process.

You must have PROCESS_VM_OPERATION access to the process. If you do not, the function fails.

lpAddress

[in] Pointer that specifies a desired starting address for the region of pages that you want to allocate.

If you are reserving memory, the function rounds this address down to the nearest 64-kilobyte boundary.

If you are committing memory that is already reserved, the function rounds this address down to the nearest page boundary. To determine the size of a page on the host computer, use the **`GetSystemInfo`** function.

If *lpAddress* is NULL, the function determines where to allocate the region.

dwSize

[in] Specifies the size, in bytes, of the region of memory to allocate.

If *lpAddress* is NULL, the function rounds *dwSize* up to the next page boundary.

If *lpAddress* is not NULL, the function allocates all pages that contain one or more bytes in the range from *lpAddress* to (*lpAddress+dwSize*). This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

flAllocationType

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

Value	Meaning
MEM_COMMIT	The function allocates actual physical storage in memory or in the paging file on disk for the specified region of memory pages. The function initializes the memory to zero. An attempt to commit a memory page that is already committed does not cause the function to fail. This means that you can commit a range of pages without first determining the current commitment state of each page. If a memory page is not yet reserved, setting this value causes the function to both reserve and commit the memory page.
MEM_RESERVE	The function reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk. Other memory allocation functions, such as **malloc** and **`LocalAlloc`**, cannot use a reserved range of memory until it is released. You can commit reserved memory pages in subsequent calls to the **VirtualAllocEx** function.

MEM_RESET	**Windows NT/2000:** Specifies that the data in the memory range specified by *lpAddress* and *dwSize* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. Using this value does not guarantee that the range operated on with MEM_RESET will contain zeroes. If you want the range to contain zeroes, decommit the memory and then recommit it. When you use MEM_RESET, the **VirtualAllocEx** function ignores the value of *fProtect*. However, you must still set *fProtect* to a valid protection value, such as PAGE_NOACCESS. **VirtualAllocEx** returns an error if you use MEM_RESET and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.
MEM_TOP_DOWN	The function allocates memory at the highest possible address.

flProtect

[in] Specifies access protection for the region of pages you are allocating. You can specify one of the following values, along with the PAGE_GUARD and PAGE_NOCACHE protection values, as desired.

Value	Meaning
PAGE_READONLY	Enables read permission to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only permission and execute permission, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write permission to the committed region of pages.
PAGE_EXECUTE	Enables execute permission to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read permission to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write permission to the committed region of pages.
PAGE_GUARD	Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages thus act as a one-shot access alarm. This value is a page protection modifier. An application uses it with one of the other page protection values, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.

PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This value is a page protection modifier, and it is only valid when used with one of the page protections other than PAGE_NOACCESS.

Return Values

If the function succeeds, the return value is the base address of the allocated region of pages.

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

Remarks

The **VirtualAllocEx** function can perform the following operations:

Commit a region of pages reserved by a previous call to the **VirtualAllocEx** function.

Reserve a region of free pages.

Reserve and commit a region of free pages.

You can use **VirtualAllocEx** to reserve a block of pages and then make additional calls to **VirtualAllocEx** to commit individual pages from the reserved block. This lets you reserve a range of a process's virtual address space without consuming physical storage until it is needed.

Each page of memory in a process's virtual address space is in one of the following states.

State	Meaning
Free	The page is not committed or reserved and is not accessible to the process. The **VirtualAllocEx** function can reserve, or simultaneously reserve and commit, a free page.
Reserved	The page is reserved. The range of addresses cannot be used by other allocation functions, but the page is not accessible and has no physical storage associated with it. The **VirtualAllocEx** function can commit a reserved page, but it cannot reserve it a second time. You can use the **`VirtualFreeEx`** function to release a reserved page in a specified process, making it a free page.
Committed	Physical storage is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. The **VirtualAllocEx** function can commit an already committed page. This means that you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. You can use the **`VirtualFreeEx`** function to decommit a committed page in a specified process, or to simultaneously decommit and free a committed page.

If the *lpAddress* parameter is not NULL, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages is allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

The PAGE_GUARD protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see Creating Guard Pages.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, HeapAlloc, VirtualFreeEx, VirtualLock, VirtualProtect, VirtualQuery

---

1.419 VirtualFree

The **VirtualFree** function releases, decommits, or releases and decommits a region of pages within the virtual address space of the calling process.

To free memory allocated in another process by the **VirtualAllocEx** function, use the **VirtualFreeEx** function.

```
VirtualFree: procedure
(
    var lpAddress:  var;
        dwSize:     SIZE_T;
        dwFreeType: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualFree@12" );
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be freed. If the *dwFreeType* parameter includes MEM_RELEASE, this parameter must be the base address returned by the **VirtualAlloc** function when the region of pages was reserved.

dwSize

[in] Specifies the size, in bytes, of the region to be freed. If the *dwFreeType* parameter includes MEM_RELEASE, this parameter must be zero. Otherwise, the region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be freed.

dwFreeType

[in] Specifies the type of free operation. This parameter can be one, or both, of the following values.

Value	Meaning
MEM_DECOMMIT	Decommits the specified region of committed pages. An attempt to decommit an uncommitted page will not cause the function to fail. This means that a range of committed or uncommitted pages can be decommitted without having to worry about a failure.
MEM_RELEASE	Releases the specified region of reserved pages. If this value is specified, the *dwSize* parameter must be zero, or the function fails.

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

VirtualFree can perform one of the following operations:

Decommit a region of committed or uncommitted pages.

Release a region of reserved pages.

Decommit and release a region of committed or uncommitted pages.

Pages that have been released are free pages available for subsequent allocation operations. Attempting to read from or write to a free page results in an access violation exception.

VirtualFree can decommit an uncommitted page; this means that a range of committed or uncommitted pages can be decommitted without having to worry about a failure. Decommitting a page releases its physical storage, either in memory or in the paging file on disk. If a page is decommitted but not released, its state changes to reserved, and it can be committed again by a subsequent call to **VirtualAlloc**. Attempting to read from or write to a reserved page results in an access violation exception.

The current state of the entire range of pages must be compatible with the type of free operation specified by the *dwFreeType* parameter. Otherwise, the function fails and no pages are released or decommitted.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, VirtualFreeEx

---

1.420 VirtualFreeEx

The **VirtualFreeEx** function releases, decommits, or releases and decommits a region of memory within the virtual address space of a specified process.

```
VirtualFreeEx: procedure
(
        hProcess:   dword;
    var lpAddress:  dword;
        dwSize:     SIZE_T;
        dwFreeType: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualFreeEx@16" );
```

Parameters

hProcess

[in] Handle to a process. The function frees memory within the virtual address space of this process.

You must have PROCESS_VM_OPERATION access to this process. If you do not, the function fails.

lpAddress

[in] Pointer to the starting address of the region of memory to free.

If the MEM_RELEASE value is used in the *dwFreeType* parameter, *lpAddress* must be the base address returned by the **VirtualAllocEx** function when the region was reserved.

dwSize

[in] Specifies the size, in bytes, of the region of memory to free.

If the MEM_RELEASE value is used in the *dwFreeType* parameter, *dwSize* must be zero. The function frees the entire region that was reserved in the initial allocation call to **VirtualAllocEx**.

If the MEM_DECOMMIT value is used, the function decommits all memory pages that contain one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means, for example, that a 2-byte region of memory that straddles a page boundary causes both pages to be decommitted.

The function decommits the entire region that was reserved by **`VirtualAllocEx`**. If the following three conditions are met:

MEM_DECOMMIT is used

lpAddress is the base address returned by the **VirtualAllocEx** function when the region was reserved

dwSize is zero

The entire region the enters the reserved state.

dwFreeType

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

Value	Meaning
MEM_DECOMMIT	The function decommits the specified region of pages. The pages enter the reserved state. The function does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining their current commitment state.
MEM_RELEASE	The function releases the specified region of pages. The pages enter the free state. If you specify this value, *dwSize* must be zero, and *lpAddress* must point to the base address returned by the **`VirtualAllocEx`** function when the region was reserved. The function fails if either of these conditions is not met. If any pages in the region are currently committed, the function first decommits and then releases them. The function does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining their current commitment state.

Return Values

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

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

Remarks

Each page of memory in a process's virtual address space is in one of the following states.

State	Meaning
Free	The page is neither committed nor reserved. The page is not accessible to the process. Attempting to read from or write to a free page results in an access violation exception. You can use the **VirtualFreeEx** function to put reserved or committed memory pages into the free state.

Reserved	The page is reserved. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. Attempting to read from or write to a free page results in an access violation exception. You can use the **VirtualFreeEx** function to put committed memory pages into the reserved state, and to put reserved memory pages into the free state.
Committed	The page is committed. Physical storage in memory or in the paging file on disk is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read from or write to that page. When a process terminates, the system releases all storage for committed pages. You can use the **`VirtualAllocEx`** function to put committed memory pages into either the reserved or free state.

The **VirtualFreeEx** function can perform the following operations:

Decommit a region of committed or uncommitted pages. After this operation, the pages are in the reserved state.

Release a region of reserved pages. After this operation, the pages are in the free state.

Decommit and release a region of committed or uncommitted pages. After this operation, the pages are in the free state.

The **VirtualFreeEx** function can decommit a range of pages that are in different states, some committed and some uncommitted. This means that you can decommit a range of pages without first determining the current commitment state of each page. Decommitting a page releases its physical storage, either in memory or in the paging file on disk.

If a page is decommitted but not released, its state changes to reserved. You can subsequently call **`VirtualAllocEx`** to commit it, or **VirtualFreeEx** to release it. Attempting to read from or write to a reserved page results in an access violation exception.

The **VirtualFreeEx** function can release a range of pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state of each page. The entire range of pages originally reserved by the **VirtualAllocEx** function must be released at the same time.

If a page is released, its state changes to free, and it is available for subsequent allocation operations. Once memory is released or decommitted, you can never refer to the memory again. Any information that may have been in that memory is gone forever. Attempting to read from or write to a free page results in an access violation exception. If you require information, do not decommit or free memory containing that information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, VirtualAllocEx

---

1.421 VirtualLock

The **VirtualLock** function locks the specified region of the process's virtual address space into physical memory, ensuring that subsequent access to the region will not incur a page fault.

```
VirtualLock: procedure
(
```

```
        var lpAddress:  var;
            dwSize:     SIZE_T
    );
        stdcall;
        returns( "eax" );
        external( "__imp__VirtualLock@8" );
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be locked.

dwSize

[in] Specifies the size, in bytes, of the region to be locked. The region of affected pages includes all pages that contain one or more bytes in the range from the *lpAddress* parameter to (*lpAddress*+*dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be locked.

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

All pages in the specified region must be committed. Memory protected with PAGE_NOACCESS cannot be locked.

Locking pages into memory may degrade the performance of the system by reducing the available RAM and forcing the system to swap out other critical pages to the paging file. By default, a process can lock a maximum of 30 pages. The default limit is intentionally small to avoid severe performance degradation. Applications that need to lock larger numbers of pages must first call the **SetProcessWorkingSetSize** function to increase their minimum and maximum working set sizes. The maximum number of pages that a process can lock is equal to the number of pages in its minimum working set minus a small overhead.

Pages that a process has locked remain in physical memory until the process unlocks them or terminates.

To unlock a region of locked pages, use the **VirtualUnlock** function. Locked pages are automatically unlocked when the process terminates.

This function is not like the **GlobalLock** or **LocalLock** function in that it does not increment a lock count and translate a handle into a pointer. There is no lock count for virtual pages, so multiple calls to the **VirtualUnlock** function are never required to unlock a region of pages.

Windows 95/98: The **VirtualLock** function is implemented as a stub that has no effect and always returns a nonzero value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, SetProcessWorkingSetSize, VirtualUnlock

---

1.422 VirtualProtect

The **VirtualProtect** function changes the access protection on a region of committed pages in the virtual address

space of the calling process.

To change the access protection of any process, use the **VirtualProtectEx** function.

```
VirtualProtect: procedure
(
    var lpAddress:      var;
        dwSize:         SIZE_T;
        flNewProtect:   dword;
    var lpflOldProtect: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualProtect@16" );
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the **VirtualAlloc** or **VirtualAllocEx** function using MEM_RESERVE. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using MEM_RESERVE.

dwSize

[in] Specifies the size, in bytes, of the region whose access protection attributes are to be changed. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

flNewProtect

[in] Specifies the new access protection. You can specify any one of the following value, along with the PAGE_GUARD and PAGE_NOCACHE values, as necessary.

Value	Meaning
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY	**Windows NT/2000:** Gives copy-on-write access to the committed region of pages.
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.

PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.
PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages thus act as a one-shot access alarm. PAGE_GUARD is a page protection modifier. An application uses it with one of the other page protection modifiers, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use PAGE_NOACCESS.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This value is a page protection modifier; it is only valid when used with one of the page protections other than PAGE_NOACCESS.

lpflOldProtect

[out] Pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

You can set the access protection value on committed pages only. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

The **VirtualProtect** function changes the access protection of memory in the calling process, and the **VirtualProtectEx** function changes the access protection of memory in a specified process.

Windows NT/2000: The PAGE_GUARD protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see Creating Guard Pages.

Windows 95/98: You cannot use **VirtualProtect** on any memory region located in the shared virtual address space (from 0x80000000 through 0xBFFFFFFF).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, VirtualAlloc, VirtualProtectEx

---

1.423 VirtualProtectEx

The **VirtualProtectEx** function changes the access protection on a region of committed pages in the virtual address space of a specified process.

```
VirtualProtectEx: procedure
(
        hProcess:       dword;
    var lpAddress:      var;
        dwSize:         SIZE_T;
        flNewProtect:   dword;
    var lpflOldProtect: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualProtectEx@20" );
```

Parameters

hProcess

[in] Handle to the process whose memory protection is to be changed. The handle must have PROCESS_VM_OPERATION access. For more information on PROCESS_VM_OPERATION, see **OpenProcess**.

lpAddress

[in] Pointer to the base address of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the **VirtualAlloc** or **VirtualAllocEx** function using MEM_RESERVE. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using MEM_RESERVE.

dwSize

[in] Specifies the size, in bytes, of the region whose access protection attributes are changed. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

flNewProtect

[in] Specifies the new access protection. You can specify any one of the following values, along with the PAGE_GUARD and PAGE_NOCACHE values, as desired.

Value	Meaning
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.

PAGE_WRITECOPY	**Windows NT/2000:** Gives copy-on-write access to the committed region of pages.
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.
PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception, and turn off the guard page status. Guard pages thus act as a one-shot access alarm. PAGE_GUARD is a page protection modifier. An application uses it with one of the other page protection modifiers, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use PAGE_NOACCESS.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be set to "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This value is a page protection modifier; it is only valid when used with one of the page protections other than PAGE_NOACCESS.

lpflOldProtect

[out] Pointer to a variable that receives the previous access protection of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

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 access protection value can be set only on committed pages. If the state of any page in the specified region is not

committed, the function fails and returns without modifying the access protection of any pages in the specified region.

VirtualProtectEx is identical to the **VirtualProtect** function except that it changes the access protection of memory in a specified process.

Windows NT/2000: The PAGE_GUARD protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see Creating Guard Pages.

Windows 95/98: You cannot use **VirtualProtectEx** on any memory region located in the shared virtual address space (from 0x80000000 through 0xBFFFFFFF).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, VirtualAlloc, VirtualProtect, VirtualQueryEx

---

1.424 VirtualQuery

The **VirtualQuery** function provides information about a range of pages in the virtual address space of the calling process.

To obtain information about a range of pages in the address space of another process, use the **VirtualQueryEx** function.

```
VirtualQuery: procedure
(
    var lpAddress:  var;
    var lpBuffer:   MEMORY_BASIC_INFORMATION;
        dwLength:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualQuery@12" );
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

lpBuffer

[out] Pointer to a **MEMORY_BASIC_INFORMATION** structure in which information about the specified page range is returned.

dwLength

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

The return value is the actual number of bytes returned in the information buffer.

Remarks

VirtualQuery provides information about a region of consecutive pages beginning at a specified address that share

the following attributes:

The state of all pages is the same with MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED, or MEM_IMAGE.

If the initial page is not free, all pages in the region are part of the same initial allocation of pages reserved by a call to the **VirtualAlloc** function.

The access of all pages is the same with PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD, or PAGE_NOCACHE.

The function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages or until it encounters a page with a nonmatching set of attributes. The function returns the attributes and the size, in bytes, of the region of pages with matching attributes. For example, if there is a 40 megabyte (MB) region of free memory, and **VirtualQuery** is called on a page that is 10 MB into the region, the function will obtain a state of MEM_FREE and a size of 30 MB.

This function reports on a region of pages in the memory of the calling process, and the **VirtualQueryEx** function reports on a region of pages in the memory of a specified process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetSystemInfo, MEMORY_BASIC_INFORMATION, VirtualQueryEx

---

1.425 VirtualQueryEx

The **VirtualQueryEx** function provides information about a range of pages within the virtual address space of a specified process.

```
VirtualQueryEx: procedure
(
        hProcess:   dword;
    var lpAddress:  var;
    var lpBuffer:   MEMORY_BASIC_INFORMATION;
        dwLength:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualQueryEx@16" );
```

Parameters

hProcess

[in] Handle to the process whose memory information is queried. The handle must have been opened with PROCESS_QUERY_INFORMATION, which enables using the handle to read information from the process object.

lpAddress

[in] Pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

lpBuffer

[out] Pointer to a **MEMORY_BASIC_INFORMATION** structure in which information about the specified page range is returned.

dwLength

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

The return value is the actual number of bytes returned in the information buffer.

Remarks

VirtualQueryEx provides information about a region of consecutive pages beginning at a specified address that share the following attributes:

The state of all pages is the same with MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED, or MEM_IMAGE.

If the initial page is not free, all pages in the region are part of the same initial allocation of pages reserved by a call to the **VirtualAllocEx** function.

The access of all pages is the same with PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD, or PAGE_NOCACHE.

The **VirtualQueryEx** function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages, or until it encounters a page with a nonmatching set of attributes. The function returns the attributes and the size, in bytes, of the region of pages with matching attributes. For example, if there is a 40 megabyte (MB) region of free memory, and **VirtualQueryEx** is called on a page that is 10 MB into the region, the function will obtain a state of MEM_FREE and a size of 30 MB.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, GetSystemInfo, MEMORY_BASIC_INFORMATION, VirtualAllocEx, VirtualProtectEx

---

1.426 VirtualUnlock

The **VirtualUnlock** function unlocks a specified range of pages in the virtual address space of a process, enabling the system to swap the pages out to the paging file if necessary.

```
VirtualUnlock: procedure
(
    var lpAddress:  var;
        dwSize:     SIZE_T
);
    stdcall;
    returns( "eax" );
    external( "__imp__VirtualUnlock@8" );
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be unlocked.

dwSize

[in] Specifies the size, in bytes, of the region being unlocked. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be unlocked.

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

For the function to succeed, the range specified need not match a range passed to a previous call to the **VirtualLock** function, but all pages in the range must be locked.

Windows NT/2000: Calling **VirtualUnlock** on a range of memory that is not locked releases the pages from the process's working set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Memory Management Overview, Memory Management Functions, VirtualLock

---

1.427 WaitCommEvent

The **WaitCommEvent** function waits for an event to occur for a specified communications device. The set of events that are monitored by this function is contained in the event mask associated with the device handle.

```
WaitCommEvent: procedure
(
        hFile:          dword;
    var lpEvtMask:      dword;
    var lpOverlapped:   OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitCommEvent@12" );
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpEvtMask

[out] Pointer to a variable that receives a mask indicating the type of event that occurred. If an error occurs, the value is zero; otherwise, it is one of the following values.

Value	Meaning
EV_BREAK	A break was detected on input.
EV_CTS	The CTS (clear-to-send) signal changed state.
EV_DSR	The DSR (data-set-ready) signal changed state.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state.
EV_RXCHAR	A character was received and placed in the input buffer.
EV_RXFLAG	The event character was received and placed in the input buffer. The event character is specified in the device's **DCB** structure, which is applied to a serial port by using the **SetCommState** function.
EV_TXEMPTY	The last character in the output buffer was sent.

lpOverlapped

[in] Pointer 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 operation is complete.

If *hFile* was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, **WaitCommEvent** is performed as an overlapped operation. In this case, the **OVERLAPPED** structure must contain a handle to a manual-reset event object (created by using the **CreateEvent** function).

If *hFile* was not opened with FILE_FLAG_OVERLAPPED, **WaitCommEvent** does not return until one of the specified events or an error occurs.

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 **WaitCommEvent** function monitors a set of events for a specified communications resource. To set and query the current event mask of a communications resource, use the **SetCommMask** and **GetCommMask** functions.

If the overlapped operation cannot be completed immediately, the function returns FALSE and the **GetLastError** function returns ERROR_IO_PENDING, indicating that the operation is executing in the background. When this happens, the system sets the **hEvent** member of the **OVERLAPPED** structure to the not-signaled state before **WaitCommEvent** returns, and then it sets it to the signaled state when one of the specified events or an error occurs. The calling process can use one of the wait functions to determine the event object's state and then use the **GetOverlappedResult** function to determine the results of the **WaitCommEvent** operation. **GetOverlappedResult** reports the success or failure of the operation, and the variable pointed to by the *lpEvtMask* parameter is set to indicate the event that occurred.

If a process attempts to change the device handle's event mask by using the **SetCommMask** function while an overlapped **WaitCommEvent** operation is in progress, **WaitCommEvent** returns immediately. The variable pointed to by the *lpEvtMask* parameter is set to zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Communications Overview, Communication Functions, CreateFile, DCB, GetCommMask, GetOverlappedResult, OVERLAPPED, SetCommMask, SetCommState

---

1.428 WaitForDebugEvent

The **WaitForDebugEvent** function waits for a debugging event to occur in a process being debugged.

```
WaitForDebugEvent: procedure
(
    var lpDebugEvent:   DEBUG_EVENT;
        dwMilliseconds: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitForDebugEvent@8" );
```

Parameters

lpDebugEvent

[in] Pointer to a **DEBUG_EVENT** structure that is filled with information about the debugging event.

dwMilliseconds

[in] Specifies the number of milliseconds to wait for a debugging event. If this parameter is zero, the function tests for a debugging event and returns immediately. If the parameter is INFINITE, the function does not return until a debugging event has occurred.

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

Only the thread that created the process being debugged can call **WaitForDebugEvent**.

When a CREATE_PROCESS_DEBUG_EVENT occurs, the debugger application receives a handle to the image file of the process being debugged, a handle to the process being debugged, and a handle to the initial thread of the process being debugged in the **DEBUG_EVENT** structure. The DEBUG_EVENT members these handles are returned in are **u.CreateProcessInfo.hFile**, **u.CreateProcessInfo.hProcess**, and **u.CreateProcessInfo.hThread** respectively. The system will close these handles. The debugger should not close these handles.

Similarly, when a CREATE_THREAD_DEBUG_EVENT occurs, the debugger application receives a handle to the thread whose creation caused the debugging event in the **uCreateThread.hThread** member of the **DEBUG_EVENT** structure. This handle should also not be closed by the debugger application, as it will be closed by the system.

Also, when a LOAD_DLL_DEBUG_EVENT occurs, the debugger application receives a handle to the loaded DLL in the **u.LoadDll.hFile** member of the **DEBUG_EVENT** structure. This handle should be closed by the debugger application by calling the **CloseHandle** function when the corresponding UNLOAD_DLL_DEBUG_EVENT occurs.

Warning Do not queue an asynchronous procedure call (APC) to a thread that calls **WaitForDebugEvent**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, ContinueDebugEvent, DebugActiveProcess, DebugBreak, DEBUG_EVENT, OutputDebugString

---

1.429 WaitForMultipleObjects

The **WaitForMultipleObjects** function returns when one of the following occurs:

Either any one or all of the specified objects are in the signaled state.

The time-out interval elapses.

To enter an alertable wait state, use the **WaitForMultipleObjectsEx** function.

```
WaitForMultipleObjects: procedure
(
        nCount:         dword;
    var lpHandles:      dword;
        fWaitAll:       boolean;
        dwMilliseconds: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitForMultipleObjects@16" );
```

Parameters

nCount

[in] Specifies the number of object handles in the array pointed to by *lpHandles*. The maximum number of object handles is MAXIMUM_WAIT_OBJECTS.

lpHandles

[in] Pointer to an array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles to objects of different types. It may not contain the multiple copies of the same handle.

If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handles must have SYNCHRONIZE access. For more information, see Standard Access Rights.

Windows 95: No handle may be a duplicate of another handle created using **DuplicateHandle**.

fWaitAll

[in] Specifies the wait type. If TRUE, the function returns when the state all objects in the *lpHandles* array is signaled. If FALSE, the function returns when the state of any one of the objects set to is signaled. In the latter case, the return value indicates the object whose state caused the function to return.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the condi-

tions specified by the *bWaitAll* parameter are not met. If *dwMilliseconds* is zero, the function tests the states of the specified objects and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

Value	Meaning
WAIT_OBJECT_0 to (WAIT_OBJECT_0 + *nCount* – 1)	If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled. If *bWaitAll* is FALSE, the return value minus WAIT_OBJECT_0 indicates the *lpHandles* array index of the object that satisfied the wait. If more than one object became signalled during the call, this is the array index of the signalled object with the smallest index value of all the signalled objects.
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + *nCount* – 1)	If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled and at least one of the objects is an abandoned mutex object. If *bWaitAll* is FALSE, the return value minus WAIT_ABANDONED_0 indicates the *lpHandles* array index of an abandoned mutex object that satisfied the wait.
WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *bWaitAll* parameter are not satisfied.

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

Remarks

The **WaitForMultipleObjects** function determines whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

When *fWaitAll* is TRUE, the function's wait operation is completed only when the states of all objects have been set to signaled. The function does not modify the states of the specified objects until the states of all objects have been set to signaled. For example, a mutex can be signaled, but the thread does not get ownership until the states of the other objects are also set to signaled. In the meantime, some other thread may get ownership of the mutex, thereby setting its state to nonsignaled.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one. When *fWaitAll* is FALSE, and multiple objects are in the signaled state, the function chooses one of the objects to satisfy the wait; the states of the objects not selected are unaffected.

The **WaitForMultipleObjects** function can specify handles of any of the following object types in the *lpHandles* array:

Change notification

Console input

Event

Job

Mutex

Process

Semaphore

Thread

Waitable timer

For more information, see Synchronization Objects.

Use caution when calling the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **`MsgWaitForMultipleObjects`** or **`MsgWaitForMultipleObjectsEx`**, rather than **WaitForMultipleObjects**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, WaitForMultipleObjectsEx

---

1.430 WaitForMultipleObjectsEx

The **WaitForMultipleObjectsEx** function returns when one of the following occurs:

Either any one or all of the specified objects are in the signaled state.

An I/O completion routine or asynchronous procedure call (APC) is queued to the thread.

The time-out interval elapses.

```
WaitForMultipleObjectsEx: procedure
(
        nCount:         dword;
    var lpHandles:      dword;
        fWaitAll:       boolean;
        dwMilliseconds: dword;
        bAlertable:     boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitForMultipleObjectsEx@20" );
```

Parameters

nCount

[in] Specifies the number of object handles to wait for in the array pointed to by *lpHandles*. The maximum number of object handles is MAXIMUM_WAIT_OBJECTS.

lpHandles

[in] Pointer to an array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles of objects of different types. It may not contain the multiple copies of the same handle.

If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handles must have SYNCHRONIZE access. For more information, see Standard Access Rights.

Windows 95: No handle may be a duplicate of another handle created using **DuplicateHandle**.

fWaitAll

[in] Specifies the wait type. If TRUE, the function returns when the states all objects in the *lpHandles* array are set to signaled. If FALSE, the function returns when the state of any one of the objects is set to signaled. In the latter case, the return value indicates the object whose state caused the function to return.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the criteria specified by the *bWaitAll* parameter are not met and no completion routines or APCs are queued. If *dwMilliseconds* is zero, the function tests the states of the specified objects and checks for queued completion routines or APCs and then returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

bAlertable

[in] Specifies whether the function returns when the system queues an I/O completion routine or APC. If TRUE, the function returns and the completion routine or APC function is executed. If FALSE, the function does not return and the completion routine or APC function is not executed.

A completion routine is queued when the **ReadFileEx** or **WriteFileEx** function in which it was specified has completed. The wait function returns and the completion routine is called only if *bAlertable* is TRUE and the calling thread is the thread that initiated the read or write operation. An APC is queued when you call **QueueUserAPC**.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

Value	Meaning
WAIT_OBJECT_0 to (WAIT_OBJECT_0 + *nCount* – 1)	If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled. If *bWaitAll* is FALSE, the return value minus WAIT_OBJECT_0 indicates the *lpHandles* array index of the object that satisfied the wait. If more than one object became signalled during the call, this is the array index of the signalled object with the smallest index value of all the signalled objects.
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + *nCount* – 1)	If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled, and at least one of the objects is an abandoned mutex object. If *bWaitAll* is FALSE, the return value minus WAIT_ABANDONED_0 indicates the *lpHandles* array index of an abandoned mutex object that satisfied the wait.
WAIT_IO_COMPLETION	One or more I/O completion routines are queued for execution.
WAIT_TIMEOUT	The time-out interval elapsed, the conditions specified by the *bWaitAll* parameter were not satisfied, and no completion routines are queued.

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

Remarks

The **WaitForMultipleObjectsEx** function determines whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

When *fWaitAll* is TRUE, the function's wait operation is completed only when the states of all objects have been set to

signaled. The function does not modify the states of the specified objects until the states of all objects have been set to signaled. For example, a mutex can be signaled, but the thread does not get ownership until the states of the other objects are also set to signaled. In the meantime, some other thread may get ownership of the mutex, thereby setting its state to nonsignaled.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one. When *fWaitAll* is FALSE, and multiple objects are in the signaled state, the function chooses one of the objects to satisfy the wait; the states of the objects not selected are unaffected.

The **WaitForMultipleObjectsEx** function can specify handles of any of the following object types in the *lpHandles* array:

Change notification

Console input

Event

Job

Mutex

Process

Semaphore

Thread

Waitable timer

For more information, see Synchronization Objects.

Use caution when calling the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **`MsgWaitForMultipleObjects`** or **`MsgWaitForMultipleObjectsEx`**, rather than **WaitForMultipleObjectsEx**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx

---

1.431 WaitForSingleObject

The **WaitForSingleObject** function returns when one of the following occurs:

The specified object is in the signaled state.

The time-out interval elapses.

To enter an alertable wait state, use the **`WaitForSingleObjectEx`** function. To wait for multiple objects, use the **`WaitForMultipleObjects`**.

```
WaitForSingleObject: procedure
(
    hHandle:        dword;
    dwMilliseconds: dword
);
```

```
stdcall;
returns( "eax" );
external( "__imp__WaitForSingleObject@8" );
```

Parameters

hHandle

[in] Handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.

If this handle is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handle must have SYNCHRONIZE access. For more information, see Standard Access Rights.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

Value	Meaning
WAIT_ABANDONED	The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to nonsignaled.
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is nonsignaled.

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

Remarks

The **WaitForSingleObject** function checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters the wait state. It uses no processor time while waiting for the object state to become signaled or the time-out interval to elapse.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one.

The **WaitForSingleObject** function can wait for the following objects:

Change notification

Console input

Event

Job

Mutex

Process

Semaphore

Thread

Waitable timer

For more information, see Synchronization Objects.

Use caution when calling the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **`MsgWaitForMultipleObjects`** or **`MsgWaitForMultipleObjectsEx`**, rather than **WaitForSingleObject**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, WaitForMultipleObjects, WaitForSingleObjectEx

---

1.432 WaitForSingleObjectEx

The **WaitForSingleObjectEx** function returns when one of the following occurs:

The specified object is in the signaled state.

An I/O completion routine or asynchronous procedure call (APC) is queued to the thread.

The time-out interval elapses.

To wait for multiple objects, use the **`WaitForMultipleObjectsEx`**.

```
WaitForSingleObjectEx: procedure
(
    hHandle:        dword;
    dwMilliseconds: dword;
    bAlertable:     boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitForSingleObjectEx@12" );
```

Parameters

hHandle

[in] Handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.

If this handle is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handle must have SYNCHRONIZE access. For more information, see Standard Access Rights.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled and no completion routines or APCs are queued. If *dwMilliseconds* is zero, the function tests the object's state and checks for queued completion routines or APCs and then returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

bAlertable

[in] Specifies whether the function returns when the system queues an I/O completion routine or APC. If TRUE,

the function returns and the completion routine or APC function is executed. If FALSE, the function does not return, and the completion routine or APC function is not executed.

A completion routine is queued when the **ReadFileEx** or **WriteFileEx** function in which it was specified has completed. The wait function returns and the completion routine is called only if *bAlertable* is TRUE, and the calling thread is the thread that initiated the read or write operation. An APC is queued when you call **QueueUserAPC**.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

Value	Meaning
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_ABANDONED	The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to nonsignaled.
WAIT_IO_COMPLETION	One or more I/O completion routines are queued for execution.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is nonsignaled.

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

Remarks

The **WaitForSingleObjectEx** function determines whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one.

The **WaitForSingleObjectEx** function can wait for the following objects:

Change notification

Console input

Event

Job

Mutex

Process

Semaphore

Thread

Waitable timer

For more information, see Synchronization Objects.

Use caution when calling the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a wait function with no time-out interval may cause the system to become deadlocked. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than **WaitForSingleObjectEx**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Synchronization Overview, Synchronization Functions, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, WaitForMultipleObjectsEx

---

1.433 WaitNamedPipe

The **WaitNamedPipe** function waits until either a time-out interval elapses or an instance of the specified named pipe is available for connection (that is, the pipe's server process has a pending **ConnectNamedPipe** operation on the pipe).

```
WaitNamedPipe: procedure
(
    lpNamedPipeName:    string;
    nTimeOut:           dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WaitNamedPipeA@8" );
```

Parameters

lpNamedPipeName

[in] Pointer to a null-terminated string that specifies the name of the named pipe. The string must include the name of the computer on which the server process is executing. A period may be used for the *servername* if the pipe is local. The following pipe name format is used:

servername\pipe*pipename*

nTimeOut

[in] Specifies the number of milliseconds that the function will wait for an instance of the named pipe to be available. You can used one of the following values instead of specifying a number of milliseconds.

Value	Meaning
NMPWAIT_USE_DEFAULT_WAIT	The time-out interval is the default value specified by the server process in the **CreateNamedPipe** function.
NMPWAIT_WAIT_FOREVER	The function does not return until an instance of the named pipe is available.

Return Values

If an instance of the pipe is available before the time-out interval elapses, the return value is nonzero.

If an instance of the pipe is not available before the time-out interval elapses, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If no instances of the specified named pipe exist, the **WaitNamedPipe** function returns immediately, regardless of the time-out value.

If the function succeeds, the process should use the **CreateFile** function to open a handle to the named pipe. A return value of TRUE indicates that there is at least one instance of the pipe available. A subsequent **CreateFile** call to the pipe can fail, because the instance was closed by the server or opened by another client.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Pipes Overview, Pipe Functions, CallNamedPipe, ConnectNamedPipe, CreateFile, CreateNamedPipe

---

1.434 WideCharToMultiByte

The **WideCharToMultiByte** function maps a wide-character string to a new character string. The new character string is not necessarily from a multibyte character set.

```
WideCharToMultiByte: procedure
(
        CodePage:           dword;
        dwFlags:            dword;
    var lpWideCharStr:      var;
        cchWideChar:        dword;
    var lpMultiByteStr:     var;
        cbMultiByte:        dword;
    var lpDefaultChar:      var;
    var lpUsedDefaultChar:  boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__WideCharToMultiByte@32" );
```

Parameters

CodePage

[in] Specifies the code page used to perform the conversion. This parameter can be given the value of any code page that is installed or available in the system. You can also specify one of the following values.

Value	Meaning
CP_ACP	ANSI code page
CP_MACCP	Macintosh code page
CP_OEMCP	OEM code page
CP_SYMBOL	**Windows 2000:** Symbol code page (42)
CP_THREAD_ACP	**Windows 2000:** Current thread's ANSI code page
CP_UTF7	**Windows NT 4.0 and Windows 2000:** Translate using UTF-7. When this is set, *lpDefaultChar* and *lpUsedDefaultChar* must be NULL
CP_UTF8	**Windows NT 4.0 and Windows 2000:** Translate using UTF-8. When this is set, *dwFlags* must be zero and both *lpDefaultChar* and *lpUsedDefaultChar* must be NULL.

dwFlags

[in] Specifies the handling of unmapped characters. The function performs more quickly when none of these

flags is set. The following flag constants are defined.

Value	Meaning
WC_NO_BEST_FIT_CHARS	**Windows 2000:** Any Unicode characters that do not translate directly to multibyte equivalents will be translated to the default character (see *lpDefaultChar* parameter). In other words, if translating from Unicode to multibyte and back to Unicode again does not yield the exact same Unicode character, the default character is used. This flag may be used by itself or in combination with the other *dwFlag* options.
WC_COMPOSITECHECK	Convert composite characters to precomposed characters.
WC_DISCARDNS	Discard nonspacing characters during conversion.
WC_SEPCHARS	Generate separate characters during conversion. This is the default conversion behavior.
WC_DEFAULTCHAR	Replace exceptions with the default character during conversion.

When WC_COMPOSITECHECK is specified, the function converts composite characters to precomposed 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/nonspacing character combination. In the character *è*, the *e* is the base character, and the accent grave mark is the nonspacing character.

When an application specifies WC_COMPOSITECHECK, it can use the last three flags in this list (WC_DISCARDNS, WC_SEPCHARS, and WC_DEFAULTCHAR) to customize the conversion to precomposed characters. These flags determine the function's behavior when there is no precomposed mapping for a base/nonspace character combination in a wide-character string. These last three flags can only be used if the WC_COMPOSITECHECK flag is set.

The function's default behavior is to generate separate characters (WC_SEPCHARS) for unmapped composite characters.

lpWideCharStr

[in] Points to the wide-character string to be converted.

cchWideChar

[in] Specifies the number of wide characters in the string pointed to by the *lpWideCharStr* parameter. If this value is –1, the string is assumed to be null-terminated and the length is calculated automatically. The length will include the null-terminator.

lpMultiByteStr

[out] Points to the buffer to receive the translated string.

cbMultiByte

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpMultiByteStr* parameter. If this value is zero, the function returns the number of bytes required for the buffer. (In this case, the *lpMultiByteStr* buffer is not used.)

lpDefaultChar

[in] Points to the character used if a wide character cannot be represented in the specified code page. If this parameter is NULL, a system default value is used. The function is faster when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL.

Note that if *CodePage* is either CP_UTF7 or CP_UTF8, this parameter must be NULL.

lpUsedDefaultChar

[in] Points to a flag that indicates whether a default character was used. The flag is set to TRUE if one or more wide characters in the source string cannot be represented in the specified code page. Otherwise, the flag is set to FALSE. This parameter may be NULL. The function is faster when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL.

Note that if *CodePage* is either CP_UTF7 or CP_UTF8, this parameter must be NULL.

Return Values

If the function succeeds, and *cbMultiByte* is nonzero, the return value is the number of bytes written to the buffer pointed to by *lpMultiByteStr*. The number includes the byte for the null terminator.

If the function succeeds, and *cbMultiByte* is zero, the return value is the required size, in bytes, 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

Remarks

The *lpMultiByteStr* and *lpWideCharStr* pointers must not be the same. If they are the same, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

If *CodePage* is CP_SYMBOL and *cbMultiByte* is less than *cchWideChar*, no characters are written to *lpMultiByte*. Otherwise, if *cbMultiByte* is less than *cchWideChar*, *cbMultiByte* characters are copied to the buffer pointed to by *lpMultiByte*.

An application can use the *lpDefaultChar* parameter to change the default character used for the conversion.

As noted earlier, the **WideCharToMultiByte** function operates most efficiently when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL. The following table shows the behavior of **WideCharToMultiByte** for the four combinations of *lpDefaultChar* and *lpUsedDefaultChar*.

lpDefaultChar	lpUsedDefaultChar	Result
NULL	NULL	No default checking. This is the most efficient way to use this function.
non-NULL	NULL	Uses the specified default character, but does not set *lpUsedDefaultChar*.
NULL	non-NULL	Uses the system default character and sets *lpUsedDefaultChar* if necessary.
non-NULL	non-NULL	Uses the specified default character and sets *lpUsedDefaultChar* if necessary.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Functions, MultiByteToWideChar

1.435 WinExec

The **WinExec** function runs the specified application.

Note This function is provided only for compatibility with 16-bit Windows. Win32-based applications should use the **CreateProcess** function.

```
WinExec: procedure
(
    lpCmdLine:  string;
    uCmdShow:   dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WinExec@8" );
```

Parameters

lpCmdLine

[in] Pointer to a null-terminated character string that contains the command line (file name plus optional parameters) for the application to be executed. If the name of the executable file in the *lpCmdLine* parameter does not contain a directory path, the system searches for the executable file in this sequence:

The directory from which the application loaded.

The current directory.

The Windows system directory. The **GetSystemDirectory** function retrieves the path of this directory.

The Windows directory. The **GetWindowsDirectory** function retrieves the path of this directory.

The directories listed in the PATH environment variable.

uCmdShow

[in] Specifies how a Windows-based application window is to be shown and is used to supply the **wShowWindow** member of the **STARTUPINFO** parameter to the **CreateProcess** function. For a list of the acceptable values, see the description of the *nCmdShow* parameter of the **ShowWindow** function. For a non-Windows — based application, the PIF file, if any, for the application determines the window state.

Return Values

If the function succeeds, the return value is greater than 31.

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

Value	Meaning
0	The system is out of memory or resources.
ERROR_BAD_FORMAT	The .exe file is invalid (non-Win32 .exe or error in .exe image).
ERROR_FILE_NOT_FOUND	The specified file was not found.
ERROR_PATH_NOT_FOUND	The specified path was not found.

Remarks

In Win32, the **WinExec** function returns when the started process calls the **GetMessage** function or a time-out limit is reached. To avoid waiting for the time out delay, call the **GetMessage** function as soon as possible in any process started by a call to **WinExec**.

Requirements

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

Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, CreateProcess, GetMessage, GetSystemDirectory, GetWindowsDirectory, LoadModule, ShowWindow

---

1.436 WriteConsole

The **WriteConsole** function writes a character string to a console screen buffer beginning at the current cursor location.

```
WriteConsole: procedure
(
        hConsoleOutput:         dword;
    var lpBuffer:               var;
        nNumberOfCharsToWrite:  dword;
    var lpNumberOfCharsWritten: dword;
    var lpReserved:             var
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteConsoleA@20" );
```

Parameters

hConsoleOutput

[in] Handle to the console screen buffer to be written to. The handle must have GENERIC_WRITE access.

lpBuffer

[in] Pointer to a buffer that contains characters to be written to the screen buffer.

nNumberOfCharsToWrite

[in] Specifies the number of characters to write.

lpNumberOfCharsWritten

[out] Pointer to a variable that receives the number of **TCHARs** actually written. For the ANSI version of this function, this is the number of bytes; for the Unicode version, this is the number of characters.

lpReserved

Reserved; must be NULL.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

WriteConsole writes characters to a console screen buffer. It behaves like the **`WriteFile`** function, except it can write in either Unicode or ANSI mode. To create an application that maintains a single set of sources compatible with both modes, use **WriteConsole** rather than **WriteFile**. Although **WriteConsole** can be used only with a console screen buffer handle, **WriteFile** can be used with other handles (such as files or pipes). **WriteConsole** fails if used with a standard handle that has been redirected to be something other than a console handle.

Although an application can use **WriteConsole** in ANSI mode to write ANSI characters, consoles do not support ANSI escape sequences. However, some Win32 functions provide equivalent functionality: for example, **SetCursorPos, SetConsoleTextAttribute,** and **GetConsoleCursorInfo**.

WriteConsole writes characters to the screen buffer at the current cursor position. The cursor position advances as characters are written. The **SetConsoleCursorPosition** function sets the current cursor position.

Characters are written using the foreground and background color attributes associated with the screen buffer. The **SetConsoleTextAttribute** function changes these colors. To determine the current color attributes and the current cursor position, use **GetConsoleScreenBufferInfo**.

All of the input modes that affect the behavior of **WriteFile** have the same effect on **WriteConsole**. To retrieve and set the output modes of a console screen buffer, use the **GetConsoleMode** and **SetConsoleMode** functions.

Windows NT/2000: This function uses either Unicode characters or ANSI characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, GetConsoleCursorInfo, GetConsoleMode, GetConsoleScreenBufferInfo, ReadConsole, SetConsoleCP, SetConsoleCursorPosition, SetConsoleMode, SetConsoleOutputCP, SetConsoleTextAttribute, SetCursorPos, WriteFile

---

1.437 WriteConsoleInput

The **WriteConsoleInput** function writes data directly to the console input buffer.

```
WriteConsoleInput: procedure
(
        hConsoleInput:              dword;
    var lpBuffer:                   INPUT_RECORD;
        nLength:                    dword;
    var lpNumberOfEventsWritten:    dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteConsoleInputA@16" );
```

Parameters

hConsoleInput

[in] Handle to the console input buffer. The handle must have GENERIC_WRITE access.

lpBuffer

[in] Pointer to an **INPUT_RECORD** buffer containing data to be written to the input buffer.

nLength

[in] Specifies the number of input records to be written.

lpNumberOfEventsWritten

[out] Pointer to a variable that receives the number of input records actually written.

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

WriteConsoleInput places input records into the input buffer behind any pending events in the buffer. The input buffer grows dynamically, if necessary, to hold as many events as are written.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, INPUT_RECORD, PeekConsoleInput, ReadConsoleInput, SetConsoleCP, SetConsoleOutputCP

1.438 WriteConsoleOutput

The **WriteConsoleOutput** function writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer. The data to be written is taken from a correspondingly sized rectangular block at a specified location in the source buffer.

```
WriteConsoleOutput: procedure
(
        hConsoleOutput: dword;
    var lpBuffer:       CHAR_INFO;
        dwBufferSize:   COORD;
        dwBufferCoord:  COORD;
    VAR lpWriteRegion:  SMALL_RECT
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteConsoleOutputA@20" );
```

Parameters

hConsoleOutput

[in] Handle to the screen buffer. The handle must have GENERIC_WRITE access.

lpBuffer

[in] Pointer to a source buffer that contains the data to be written to the screen buffer. This pointer is treated as the origin of a two-dimensional array of **CHAR_INFO** structures whose size is specified by the *dwBufferSize* parameter.

dwBufferSize

[in] Specifies the size, in character cells, of the buffer pointed to by the *lpBuffer* parameter. The **X** member of the

COORD structure is the number of columns; the **Y** member is the number of rows.

dwBufferCoord

[in] Specifies the coordinates of the upper-left cell in the buffer pointed to by the *lpBuffer* parameter to write data from. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpWriteRegion

[in/out] Pointer to a **SMALL_RECT** structure. On input, the structure members specify the upper-left and lower-right coordinates of the screen buffer rectangle to write to. On output, the structure members specify the actual rectangle that was written to.

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

WriteConsoleOutput treats the source buffer and the destination screen buffer as two-dimensional arrays (columns and rows of character cells). The rectangle pointed to by the *lpWriteRegion* parameter specifies the size and location of the block to be written to in the screen buffer. A rectangle of the same size is located with its upper-left cell at the coordinates of the *dwBufferCoord* parameter in the *lpBuffer* array. Data from the cells that are in the intersection of this rectangle and the source buffer rectangle (whose dimensions are specified by the *dwBufferSize* parameter) is written to the destination rectangle.

Cells in the destination rectangle whose corresponding source location are outside the boundaries of the source buffer rectangle are left unaffected by the write operation. In other words, these are the cells for which no data is available to be written.

Before **WriteConsoleOutput** returns, it sets the members of *lpWriteRegion* to the actual screen buffer rectangle affected by the write operation. This rectangle reflects the cells in the destination rectangle for which there existed a corresponding cell in the source buffer, because **WriteConsoleOutput** clips the dimensions of the destination rectangle to the boundaries of the screen buffer.

If the rectangle specified by *lpWriteRegion* lies completely outside the boundaries of the screen buffer, or if the corresponding rectangle is positioned completely outside the boundaries of the source buffer, no data is written. In this case, the function returns with the members of the structure pointed to by the *lpWriteRegion* parameter set such that the **Right** member is less than the **Left**, or the **Bottom** member is less than the **Top**. To determine the size of the screen buffer, use the **GetConsoleScreenBufferInfo** function.

WriteConsoleOutput has no effect on the cursor position.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, CHAR_INFO, COORD, GetConsoleScreenBufferInfo, ReadConsoleOutput, ReadConsoleOutputAttribute, ReadConsoleOutputCharacter, SetConsoleCP, SetConsoleOutputCP, SMALL_RECT, WriteConsoleOutputAttribute, WriteConsoleOutputCharacter

1.439 WriteConsoleOutputAttribute

The **WriteConsoleOutputAttribute** function copies a number of foreground and background color attributes to consecutive cells of a console screen buffer, beginning at a specified location.

```
WriteConsoleOutputAttribute: procedure
(
        hConsoleOutput:         dword;
    var lpAttribute:            word;
        nLength:                dword;
        dwWriteCoord:           COORD;
    var lpNumberOfAttrsWritten: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteConsoleOutputAttribute@20" );
```

Parameters

hConsoleOutput

[in] Handle to the screen buffer. The handle must have GENERIC_WRITE access.

lpAttribute

[in] Pointer to a buffer that contains the attributes to write to the screen buffer.

nLength

[in] Specifies the number of screen buffer character cells to write to.

dwWriteCoord

[in] Specifies the column and row coordinates of the first cell in the screen buffer to write to.

lpNumberOfAttrsWritten

[out] Pointer to a variable that receives the number of attributes actually written to the screen buffer.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of attributes to be written to extends beyond the end of the specified row in the screen buffer, attributes are written to the next row. If the number of attributes to be written to extends beyond the end of the screen buffer, the attributes are written up to the end of the screen buffer.

The character values at the positions written to are not changed.

Each attribute specifies the foreground (text) and background colors in which that character cell is drawn. The attribute values are some combination of the following values: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces red text on a white background:

```
FOREGROUND_RED | BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf

Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, ReadConsoleOutput, ReadConsoleOutputAttribute, ReadConsoleOutputCharacter, WriteConsoleOutput, WriteConsoleOutputCharacter

---

1.440 WriteConsoleOutputCharacter

The **WriteConsoleOutputCharacter** function copies a number of characters to consecutive cells of a console screen buffer, beginning at a specified location.

```
WriteConsoleOutputCharacter: procedure
(
        hConsoleOutput:         dword;
        lpCharacter:            string;
        nLength:                dword;
        dwWriteCoord:           COORD;
    var lpNumberOfCharsWritten: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteConsoleOutputCharacterA@20" );
```

Parameters

hConsoleOutput

[in] Handle to the screen buffer. The handle must have GENERIC_WRITE access.

lpCharacter

[in] Pointer to a buffer that contains the characters to write to the screen buffer.

nLength

[in] Specifies the number of **TCHARs** to write. For the ANSI version of the function, this is the number of bytes; for the Unicode version, this is the number of characters.

dwWriteCoord

[in] Specifies the column and row coordinates of the first cell in the screen buffer to write to.

lpNumberOfCharsWritten

[out] Pointer to a variable that receives the number of **TCHARs** actually written. For the ANSI version of the function, this is the number of bytes; for the Unicode version, this is the number of characters.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

If the number of characters to be written to extends beyond the end of the specified row in the screen buffer, characters are written to the next row. If the number of characters to be written to extends beyond the end of the screen buffer, characters are written up to the end of the screen buffer.

The attribute values at the positions written to are not changed.

Windows NT/2000: This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page,

use the **SetConsoleCP** or **SetConsoleOutputCP** functions, or use the **chcp** or **mode con cp select=** commands.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Consoles and Character-Mode Support Overview, Console Functions, ReadConsoleOutput, ReadConsoleOutputAttribute, ReadConsoleOutputCharacter, SetConsoleCP, SetConsoleOutputCP, WriteConsoleOutput, WriteConsoleOutputAttribute

---

1.441 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.

This function is designed for both synchronous and asynchronous operation. The **WriteFileEx** function is designed solely for asynchronous operation. It lets an application perform other processing during a file write operation.

```
WriteFile: procedure
(
        hFile:                  dword;
    var lpBuffer:               var;
        nNumberOfBytesToWrite:  dword;
    var lpNumberOfBytesWritten: dword;
    var lpOverlapped:           OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteFile@20" );
```

Parameters

hFile

[in] Handle to the file to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

Windows NT/2000: 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** function.

Windows 95/98: For asynchronous write operations, *hFile* can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous write operations on mailslots, named pipes, or disk files.

lpBuffer

[in] Pointer to the buffer containing the data to be written to the file.

nNumberOfBytesToWrite

[in] Specifies the number of bytes to write to the file.

A value of zero specifies 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 65,535 bytes.

lpNumberOfBytesWritten

[out] Pointer to the variable that receives the number of bytes written. **WriteFile** sets this value to zero before doing any work or error checking.

Windows NT/2000: 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**.

Windows 95/98: This parameter cannot be NULL.

lpOverlapped

[in] Pointer 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.

Windows NT/2000: 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.

Windows 95/98: For operations on files, disks, pipes, or mailslots, this parameter must be NULL; a pointer to an **OVERLAPPED** structure causes the call to fail. However, Windows 95/98 supports overlapped I/O on serial and parallel ports.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

An application 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. To determine a volume's sector size, call the **GetDiskFreeSpace** function.

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 sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

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

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not read from, write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

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.

The system 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.

MAPI: For more information, see Syntax and Limitations for Win32 Functions Useful in MAPI Development.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CancelIo, CreateFile, GetLastError, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED, ReadFile, SetEndOfFile, SetErrorMode, WriteFileEx

---

1.442 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 or canceled and the calling thread is in an alertable wait state.

```
WriteFileEx: procedure
(
        hFile:                  dword;
    var lpBuffer:               var;
        nNumberOfBytesToWrite:  dword;
    var lpOverlapped:           OVERLAPPED;
        lpCompletionRoutine:    procedure
);
    stdcall;
    returns( "eax" );
```

```
external( "__imp__WriteFileEx@20" );
```

Parameters

hFile

[in] Handle to the file 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/ 2000: This parameter 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`** function.

Windows 95/98: This parameter can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous write operations on mailslots, named pipes, or disk files.

lpBuffer

[in] Pointer 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

[in] 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

[in] Pointer 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, **Offset** and **OffsetHigh** are ignored.

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 queuing 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

[in] Pointer 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

When using **WriteFileEx** you should check **GetLastError** even when the function returns "success" to check for conditions that are "successes" but have some outcome you might want to know about. For example, a buffer overflow when calling **WriteFileEx** will return TRUE, but **GetLastError** will report the overflow with ERROR_MORE_DATA. If the function call is successful and there are no warning conditions, **GetLastError** will return ERROR_SUCCESS.

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. To determine a volume's sector size, call the **GetDiskFreeSpace** function.

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 sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

If part of the output file is locked by another process, and the specified write operation overlaps the locked portion, the **WriteFileEx** function fails.

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not read from, write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

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.

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 95/98: 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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CancelIo, CreateFile, FileIOCompletionRoutine, MsgWaitForMultipleObjectsEx, OVERLAPPED, ReadFileEx, SetEndOfFile, SetErrorMode, SleepEx, SignalObjectAndWait, WaitForMultipleObjectsEx, WaitForSingleObjectEx, WriteFile

1.443 WriteFileGather

The **WriteFileGather** function gathers data from a set of buffers and writes the data to a file.

The function starts writing data to the file at a position specified by an **OVERLAPPED** structure. It operates asynchronously.

```
WriteFileGather: procedure
(
        hFile:                  dword;
    var aSegmentArray:          FILE_SEGMENT_ELEMENT;
        nNumberOfBytesToWrite:  dword;
    var lpReserved:             dword;
    var lpOverlapped:           OVERLAPPED
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteFileGather@20" );
```

Parameters

hFile

[in] Handle to the file to write to.

This file handle must have been created using GENERIC_WRITE to specify write access to the file, FILE_FLAG_OVERLAPPED to specify asynchronous I/O, and FILE_FLAG_NO_BUFFERING to specify non-cached I/O.

aSegmentArray

[out] Pointer to an array of **FILE_SEGMENT_ELEMENT** pointers to buffers. The function gathers the data it writes to the file from this set of buffers.

Each buffer must be at least the size of a system memory page and must be aligned on a system memory page size boundary. The system will write one system memory page of data from each buffer that a **FILE_SEGMENT_ELEMENT** pointer points to.

The function gathers the data from the buffers in a sequential manner: it writes data to the file from the first buffer, then from the second buffer, then from the next, until there is no more data to write.

The final element of the array must be a 64-bit NULL pointer.

Note The array must contain one member for each system memory page-sized chunk of the total number of bytes to write to the file, plus one member for the final NULL pointer. For example, if the number of bytes to read is 40K, and the system page size is 4K, then this array must have 10 members for data, plus one member for the final NULL member, for a total of 11 members.

nNumberOfBytesToWrite

[in] Specifies the total number of bytes to write; each element of *aSegmentArray* contains a 1-page chunk of this total. Because the file must be opened with FILE_FLAG_NO_BUFFERING, the number of bytes to write must be a multiple of the sector size of the file system on which the file resides.

If *nNumberOfBytesToWrite* is zero, the function performs a null write operation. A null write operation does not write any bytes to the file, but it does cause the file's time stamp to change.

Note that this behavior differs from file writing functions on the MS-DOS platform, where a write count of zero bytes truncates a file. If *nNumberOfBytesToWrite* is zero, **WriteFileGather** does not truncate or extend the file. To truncate or extend a file, use the **SetEndOfFile** function. However, if *nNumberOfBytesToWrite* is not zero and the offset and length of the write place data beyond the current end of the file, **WriteFileGather** will extend the file.

lpReserved

This parameter is reserved for future use. You must set it to NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** data structure.

The **WriteFileGather** function requires a valid **OVERLAPPED** structure. The *lpOverlapped* parameter cannot be NULL.

The **WriteFileGather** function starts writing data to the file at a position specified by the **Offset** and **OffsetHigh** members of the **OVERLAPPED** structure.

The **WriteFileGather** function may return before the write operation has completed. In that case, the **WriteFileGather** function returns the value zero, and the **GetLastError** function returns the value ERROR_IO_PENDING. This asynchronous operation of **WriteFileGather** lets the calling process continue while the write operation completes. You can call the **GetOverlappedResult, HasOverlappedIoCompleted,** or **GetQueuedCompletionStatus** function to obtain information about the completion of the write operation.

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 the **GetLastError** function.

If the function returns before the write operation has completed, the function returns zero, and **GetLastError** returns ERROR_IO_PENDING.

Remarks

If part of the file specified by *hFile* is locked by another process, and the write operation overlaps the locked portion, the **WriteFileGather** function fails.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP2 or later.
Windows 95/98: Unsupported.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

File I/O Overview, File I/O Functions, CreateFile, GetOverlappedResult, GetQueuedCompletionStatus, HasOverlappedIoCompleted, OVERLAPPED, ReadFile, ReadFileEx, ReadFileScatter

---

1.444 WritePrivateProfileSection

The **WritePrivateProfileSection** function replaces the keys and values for the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should store initialization information in the registry.

```
WritePrivateProfileSection: procedure
(
    lpAppName:  string;
    lpString:   string;
    lpFileName: string
);
    stdcall;
    returns( "eax" );
    external( "__imp__WritePrivateProfileSectionA@12" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string specifying the name of the section in which data is written. This section name is typically the name of the calling application.

lpString

[in] Pointer to a buffer containing the new key names and associated values that are to be written to the named section.

lpFileName

[in] Pointer to a null-terminated string containing the name of the initialization file. If this parameter does not contain a full path for the file, the function searches the Windows directory for the file. If the file does not exist and *lpFileName* does not contain a full path, the function creates the file in the Windows directory. The function does not create a file if *lpFileName* contains the full path and file name of a file that does not exist.

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

key=string

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

If no section name matches the string pointed to by the *lpAppName* parameter, **WritePrivateProfileSection** creates the section at the end of the specified initialization file and initializes the new section with the specified key name and value pairs.

WritePrivateProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by the *lpString* parameter. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no operations that read from or write to the specified initialization file are allowed while the information is being written.

Windows 95/98: The system 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.

Windows NT/2000: The system 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

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the **WritePrivateProfileSection** function writes information to the registry, not to 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileSection, RegCreateKeyEx, RegSetValueEx, WriteProfileSection

---

1.445 WritePrivateProfileString

The **WritePrivateProfileString** function copies a string into the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should store initialization information in the registry.

```
WritePrivateProfileString: procedure
(
    lpAppName:      string;
    lpKeyName:      string;
    lpString:       string;
    lpFileName:     string
);
    stdcall;
    returns( "eax" );
    external( "__imp__WritePrivateProfileStringA@16" );
```

Parameters

lpAppName

[in] Pointer 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

[in] Pointer 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

[in] Pointer 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: The system does not support the use of the TAB (\t) character as part of this parameter.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of 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 file name 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 file name and the file does not exist, **WriteProfileString** creates the file. The specified directory must already exist.

Windows NT/2000: The system 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/Windows 2000 keeps a cache for the IniFileMapping registry key. Calling **WritePrivateProfileStringW** with the value of all arguments set to NULL will cause the system 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 initial-

ization information:

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

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.

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.

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.

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.

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 file name. 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 system rereads 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 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);

}
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileString, WriteProfileString

---

1.446 WritePrivateProfileStruct

The **WritePrivateProfileStruct** function copies data into a key in the specified section of an initialization file. As it copies the data, the function calculates a checksum and appends it to the end of the data. The **GetPrivateProfile-Struct** function uses the checksum to ensure the integrity of the data.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should store initialization information in the registry.

```
    WritePrivateProfileStruct: procedure
    (
            lpszSection:    string;
            lpszKey:        string;
```

```
        var lpStruct:       var;
            uSizeStruct:    dword;
            szFile:         string
    );
        stdcall;
        returns( "eax" );
        external( "__imp__WritePrivateProfileStructA@20" );
```

Parameters

lpszSection

[in] Pointer 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.

lpszKey

[in] Pointer 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 keys and entries within the section, is deleted.

lpStruct

[in] Pointer to a buffer that contains the data to copy. If this parameter is NULL, the given key is deleted.

uSizeStruct

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpStruct* parameter.

szFile

[in] Pointer to a null-terminated string that names the initialization file. If this parameter is NULL, the given information is copied into the Win.ini 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 *szFile* parameter does not contain a full path and file name 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 *szFile* contains a full path and file name and the file does not exist, **WriteProfileString** creates the file. The specified directory must already exist.

Windows NT/2000: The Win32 profile functions (**Get/WriteProfile***, **Get/WritePrivateProfile***) use the following steps to locate initialization information:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetPrivateProfileString, WriteProfileString

---

1.447 WriteProcessMemory

The **WriteProcessMemory** function writes data to an area of memory in a specified process. The entire area to be written to must be accessible, or the operation fails.

```
WriteProcessMemory: procedure
(
        hProcess:               dword;
    var lpBaseAddress:          var;
    var lpBuffer:               var;
        nSize:                  dword;
    var lpNumberOfBytesWritten: dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteProcessMemory@20" );
```

Parameters

hProcess

[in] Handle to the process whose memory is to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

lpBaseAddress

[in] Pointer to the base address in the specified process to which data will be written. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access. If this is the case, the function proceeds; otherwise, the function fails.

lpBuffer

[in] Pointer to the buffer that contains data to be written into the address space of the specified process.

nSize

[in] Specifies the requested number of bytes to write into the specified process.

lpNumberOfBytesWritten

[out] Pointer to a variable that receives the number of bytes transferred into the specified process. This parameter is optional. If *lpNumberOfBytesWritten* is NULL, 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**. The function will fail if the requested write operation crosses into an area of the process that is inaccessible.

Remarks

WriteProcessMemory copies the data from the specified buffer in the current process to the address range of the specified process. Any process that has a handle with PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process to be written to can call the function. The process whose address space is being written to is typically, but not necessarily, being debugged.

The entire area to be written to must be accessible. If it is not, the function fails as noted previously.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Debugging Overview, Debugging Functions, ReadProcessMemory

1.448 WriteProfileSection

The **WriteProfileSection** function replaces the contents of the specified section in the Win.ini file with specified keys and values.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should store initialization information in the registry.

```
WriteProfileSection: procedure
(
```

```
        lpAppName:  string;
        lpString:   string
    );
        stdcall;
        returns( "eax" );
        external( "__imp__WriteProfileSectionA@8" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string containing the name of the section. This section name is typically the name of the calling application.

lpString

[in] Pointer to a buffer containing the new key names and associated values that are to be written to the named section.

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: If there is no section in Win.ini that matches the specified section name, **WriteProfileSection** creates the section at the end of the file and initializes the new section with the key name and value pairs specified in the *lpString* parameter.

Keys and values in the *lpString* buffer consist of one or more null-terminated strings, followed by a final null character. Each string has the following form:

```
key=string
```

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

WriteProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by *lpString*. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no other operations that read from or write to the initialization file are allowed while the information is being written.

Windows NT/2000: The system 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

When the operation has been mapped, the **WriteProfileSection** function writes information to the registry, not to 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetProfileSection, WritePrivateProfileSection

---

1.449 WriteProfileString

The **WriteProfileString** function copies a string into the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should store initialization information in the registry.

```
WriteProfileString: procedure
(
    lpAppName:  string;
    lpKeyName:  string;
    lpString:   string
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteProfileStringA@12" );
```

Parameters

lpAppName

[in] Pointer to a null-terminated string that specifies the section to which the string is to be copied. If the section does not exist, it is created. The name of the section is not case-sensitive; the string can be any combination of

uppercase and lowercase letters.

lpKeyName

[in] Pointer to a null-terminated string containing the key to be associated with the 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 in the section, is deleted.

lpString

[in] Pointer 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: The system does not support the use of the TAB (\t) character as part of this parameter.

Return Values

If the function successfully copies the string to the Win.ini file, the return value is nonzero.

If the function fails, or if it flushes the cached version of Win.ini, 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 Win.ini file must have the following form:

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

Windows NT/2000: The system 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

When the operation has been mapped, the **WriteProfileString** function writes information to the registry, not to 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:

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

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.

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.

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.

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.

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.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Registry Overview, Registry Functions, GetProfileString, WritePrivateProfileString

---

1.450 WriteTapemark

The **WriteTapemark** function writes a specified number of filemarks, setmarks, short filemarks, or long filemarks to a tape device. These tapemarks divide a tape partition into smaller areas.

```
WriteTapemark: procedure
(
    hDevice:            dword;
    dwTapemarkType:     dword;
    dwTapemarkCount:    dword;
    bImmediate:         boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__WriteTapemark@16" );
```

Parameters

hDevice

[in] Handle to the device on which to write tapemarks. This handle is created by using the **CreateFile** function.

dwTapemarkType

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

Value	Description
TAPE_FILEMARKS	Writes the number of filemarks specified by the *dwTapemark-Count* parameter.

TAPE_LONG_FILEMARKS	Writes the number of long filemarks specified by *dwTapemarkCount*.
TAPE_SETMARKS	Writes the number of setmarks specified by *dwTapemarkCount*.
TAPE_SHORT_FILEMARKS	Writes the number of short filemarks specified by *dwTapemarkCount*.

dwTapemarkCount

[in] Specifies the number of tapemarks to write.

bImmediate

[in] Specifies whether to return as soon as the operation begins. If this parameter is TRUE, the function returns immediately; if it is FALSE, the function does not return until the operation has been completed.

Return Values

If the function succeeds, the return value is NO_ERROR.

If the function fails, the return value is one of the following error codes:

Error	Description
ERROR_BEGINNING_OF_MEDIA	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED	The partition information could not be found when a tape was being loaded.
ERROR_MEDIA_CHANGED	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE	There is no media in the drive.
ERROR_NOT_SUPPORTED	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

Filemarks, setmarks, short filemarks, and long filemarks are special recorded elements that denote the linear organization of the tape. None of these marks contain user data. Filemarks are the most general marks; setmarks provide a hierarchy not available with filemarks.

A short filemark contains a short erase gap that cannot be overwritten unless the write operation is performed from the beginning of the partition or from an earlier long filemark.

A long filemark contains a long erase gap that allows an application to position the tape at the beginning of the filemark and to overwrite the filemark and the erase gap.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Header: Declared in kernel32.hhf
Library: Use Kernel32.lib.

See Also

Tape Backup Overview, Tape Backup Functions, CreateFile