Ayuda sobre Funciones de la API Win32

Platform SDK: DLLs, Processes, and Threads

CreateProcess

The CreateProcess function creates a new process and its primary thread. The new process runs the specified executable file in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

lpApplicationName

[in] Pointer to a null-terminated string that specifies the module to execute. The specified module can be a Windows-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.

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. If the file name does not contain an extension, .exe is assumed. Therefore, if the file name extension is .com, this parameter must include the .com extension.

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

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.

To run a batch file, you must start the command interpreter; set *lpApplicationName* to cmd.exe and set *lpCommandLine* to the name of the batch file.

lpCommandLine

[in, out] Pointer to a null-terminated string that specifies the command line to execute. The maximum length of this string is 32K characters.

> **Windows 2000: The maximum length of this string is MAX_PATH characters.**

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, the null-terminated string pointed to by *lpApplicationName* specifies the module to execute, and the null-terminated string pointed to by *lpCommandLine* specifies the command line. The new process can use GetCommandLine to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because argv[0] is the module name, C programmers generally 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. Therefore, if the file name extension is .com, this parameter must include the .com extension. 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:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory.

   Windows Me/98/95: The Windows system directory. Use the GetSystemDirectory function to get the path of this directory.
4. 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.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

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.

The lpSecurityDescriptor member of the structure specifies a security descriptor for the new process. If *lpProcessAttributes* is NULL or lpSecurityDescriptor is NULL, the process gets a default security descriptor. The ACLs in the default security descriptor for a process come from the primary token of the creator.

Windows XP/2000/NT: The ACLs in the default security descriptor for a process come from the primary or impersonation token of the creator. This behavior changed with Windows XP SP2 and Windows Server 2003.

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.

The lpSecurityDescriptor member of the structure specifies a security descriptor for the main thread. If *lpThreadAttributes* is NULL or lpSecurityDescriptor is NULL, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the process token.

Windows XP/2000/NT: The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator. This behavior changed with Windows XP SP2 and Windows Server 2003.

bInheritHandles

[in] If this parameter TRUE, each inheritable handle in the calling process is inherited by the new process. If the parameter is FALSE, the handles are not inherited. Note that inherited handles have the same value and access rights as the original handles.

dwCreationFlags

[in] Flags that control the priority class and the creation of the process. For a list of values, see Process Creation Flags.

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see GetPriorityClass. If none of the 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.

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.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes CREATE_UNICODE_ENVIRONMENT.

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 full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is NULL, the new process will have the same current drive and directory as the calling process. (This feature 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 the window station, desktop, standard handles, and appearance of the main window for the new process.

lpProcessInformation

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

Handles in PROCESS_INFORMATION must be closed with CloseHandle when they are 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 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. It can be specified in the OpenThread function to open a handle to the thread. 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.

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.

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). For more information, see Terminating a Process.

A parent process can directly alter the environment variables of a child process during process creation. This is the only situation when a process can directly change the environment settings of another process. For more information, see Changing Environment Variables.

If an application provides an environment block, the current directory information of the system drives is not automatically propagated to the new process. For example, there is an environment variable named =C: whose value is the current directory on drive C. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create these environment variable strings, sort them alphabetically (because the system uses a sorted environment), and put them into the environment block. Typically, they will go at the front of the environment block, due to the environment block sort order.

One way to obtain the current directory information 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.

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 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.

Security Remarks

The first parameter, *lpApplicationName*, can be NULL, in which case the executable name must be in the white space-delimited string pointed to by *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

```
CreateProcess(NULL, "C:\\Program Files\\MyApp", ...)
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls CreateProcess using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass NULL for *lpApplicationName*. If you do pass NULL for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

```
CreateProcess(NULL, "\"C:\\Program Files\\MyApp.exe\" -L -S", ...)
```

Example Code

For an example, see Creating Processes.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**
Unicode	**Implemented as CreateProcessW (Unicode) and CreateProcessA (ANSI). Note that Unicode support on Windows Me/98/95 requires Microsoft Layer for Unicode.**

See Also

CloseHandle, CreateProcessAsUser, CreateProcessWithLogonW, ExitProcess, GetCommandLine, GetEnvironmentStrings, GetExitCodeProcess, **GetFullPathName**, GetStartupInfo, OpenProcess, Process and Thread Functions, PROCESS_INFORMATION, Processes, **SECURITY_ATTRIBUTES**, **SetErrorMode**, STARTUPINFO, TerminateProcess, WaitForInputIdle

---

Platform SDK: DLLs, Processes, and Threads

ExitProcess

The ExitProcess function ends a process and all its threads.

```
VOID ExitProcess(
  UINT uExitCode
);
```

Parameters

uExitCode
[in] Exit code for the process and 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

This function does not return a value.

Remarks

Exiting a process causes the following:

1. **All of the object handles opened by the process are closed.**
2. **All of the threads in the process, except the calling thread, terminate their execution. The entry-point functions of all loaded dynamic-link libraries (DLLs) are called with DLL_PROCESS_DETACH. After all attached DLLs have executed any process termination code, this function terminates the current process, including the calling thread.**
3. **The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.**
4. **The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.**
5. **The termination status of the process changes from STILL_ACTIVE to the exit value of the process.**

If one of the terminated threads in the process holds a lock and the DLL detach code in one of the loaded DLLs attempts to acquire the same lock, then calling ExitProcess results in a deadlock. In contrast, if a process terminates by calling TerminateProcess, the DLLs that the process is attached to are not notified of the process termination. Therefore, if you do not know the state of all threads in your process, it is better to call TerminateProcess than ExitProcess. Note that returning from the main function of an application results in a call to ExitProcess.

Calling ExitProcess in a DLL can lead to unexpected application or system errors. Be sure to call ExitProcess from a DLL only if you know which applications or system components will load the DLL and that it is safe to call ExitProcess in this context.

Exiting a process does not cause child processes to be terminated.

Exiting 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.

Example Code

For an example, see Creating a Child Process with Redirected Input and Output.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**

See Also

CreateProcess, CreateRemoteThread, CreateThread, ExitThread, GetExitCodeProcess, GetExitCodeThread, OpenProcess, Process and Thread Functions, Processes, TerminateProcess, Terminating a Process

Platform SDK: DLLs, Processes, and Threads

Sleep

The Sleep function suspends the execution of the current thread for at least the specified interval.

To enter an alertable wait state, use the SleepEx function.

```
VOID Sleep(
  DWORD dwMilliseconds
);
```

Parameters

dwMilliseconds

[in] Minimum time interval for which execution is to be suspended, in milliseconds.

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 indicates that the suspension should not time out.

Return Values

This function does not return a value.

Remarks

This function causes a thread to relinquish the remainder of its time slice and become unrunnable for at least the specified number of milliseconds, after which the thread is ready to run. In particular, if you specify zero milliseconds, the thread will relinquish the remainder of its time slice but remain ready. Note that a ready thread is not guaranteed to run immediately. Consequently, the thread may not run until some time after the specified interval elapses. For more information, see Scheduling Priorities.

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.

Example Code

For an example, see Using Thread Local Storage.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**

See Also

MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, Process and Thread Functions, SleepEx, Suspending Thread Execution, Threads

---

MessageBox Function

The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

Syntax

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

Parameters

hWnd
[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText
[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption
[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title Error is used.

uType
[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values.

`MB_ABORTRETRYIGNORE`
The message box contains three push buttons: Abort, Retry, and Ignore.

`MB_CANCELTRYCONTINUE`
Microsoft Windows 2000/XP: The message box contains three push buttons: Cancel, Try Again, Continue. Use this message box type instead of MB_ABORTRETRYIGNORE.

`MB_HELP`
Windows 95/98/Me, Windows NT 4.0 and later: Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner.

`MB_OK`
The message box contains one push button: OK. This is the default.

`MB_OKCANCEL`
The message box contains two push buttons: OK and Cancel.

`MB_RETRYCANCEL`
The message box contains two push buttons: Retry and Cancel.

`MB_YESNO`
The message box contains two push buttons: Yes and No.

`MB_YESNOCANCEL`
The message box contains three push buttons: Yes, No, and Cancel.

To display an icon in the message box, specify one of the following values.

`MB_ICONEXCLAMATION`
An exclamation-point icon appears in the message box.

`MB_ICONWARNING`
An exclamation-point icon appears in the message box.

`MB_ICONINFORMATION`
An icon consisting of a lowercase letter *i* in a circle appears in the message box.

`MB_ICONASTERISK`
An icon consisting of a lowercase letter *i* in a circle appears in the message box.

`MB_ICONQUESTION`
A question-mark icon appears in the message box. The question mark message icon is no longer recommended because it does not clearly represent a specific type of message and because the phrasing of a message as a question could apply to any message type. In addition, users can confuse the message symbol question mark with Help information. Therefore, do not use this question mark message symbol in your message boxes. The

system continues to support its inclusion only for backward compatibility.

MB_ICONSTOP

A stop-sign icon appears in the message box.

MB_ICONERROR

A stop-sign icon appears in the message box.

MB_ICONHAND

A stop-sign icon appears in the message box.

To indicate the default button, specify one of the following values.

MB_DEFBUTTON1

The first button is the default button.

MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.

MB_DEFBUTTON2

The second button is the default button.

MB_DEFBUTTON3

The third button is the default button.

MB_DEFBUTTON4

The fourth button is the default button.

To indicate the modality of the dialog box, specify one of the following values.

MB_APPLMODAL

The user must respond to the message box before continuing work in the window identified by the *hWnd* parameter. However, the user can move to the windows of other threads and work in those windows.

Depending on the hierarchy of windows in the application, the user may be able to move to other windows within the thread. All child windows of the parent of the message box are automatically disabled, but popup windows are not.

MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.

MB_SYSTEMMODAL

Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with *hWnd*.

MB_TASKMODAL

Same as MB_APPLMODAL except that all the top-level windows belonging to the current thread are disabled if the *hWnd* parameter is NULL. Use this flag when the calling application or library does not have a window handle available but still needs to prevent input to other windows in the calling thread without suspending other threads.

To specify other options, use one or more of the following values.

MB_DEFAULT_DESKTOP_ONLY

Windows NT/2000/XP: Same as MB_SERVICE_NOTIFICATION except that the system will display the message box only on the default desktop of the interactive window station. For more information, see Window Stations.

Windows NT 4.0 and earlier: If the current input desktop is not the default desktop, MessageBox fails.

Windows 2000/XP: If the current input desktop is not the default desktop, MessageBox does not return until the user switches to the default desktop.

Windows 95/98/Me: This flag has no effect.

MB_RIGHT

The text is right-justified.

MB_RTLREADING

Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems.

MB_SETFOREGROUND

The message box becomes the foreground window. Internally, the system calls the SetForegroundWindow function for the message box.

MB_TOPMOST

The message box is created with the WS_EX_TOPMOST window style.

MB_SERVICE_NOTIFICATION

Windows NT/2000/XP: The caller is a service notifying the user of an

event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer.

Terminal Services: If the calling thread has an impersonation token, the function directs the message box to the session specified in the impersonation token.

If this flag is set, the *hWnd* parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the *hWnd*.

For more information on the changes between Microsoft Windows NT 3.51 and Windows NT 4.0, see Remarks.

MB_SERVICE_NOTIFICATION_NT3X

Windows NT/2000/XP: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51.

For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.

Return Value

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

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

If the function succeeds, the return value is one of the following menu-item values.

IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDTRYAGAIN	Try Again button was selected.
IDYES	Yes button was selected.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the *lpText* and *lpCaption* parameters should not be taken from a resource file, because an attempt to load the resource may fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the *hWnd* parameter. The *hWnd* parameter should not identify a child window, such as a control in a dialog box.

Windows 95/98/Me: The system can support a maximum of 16,364 window handles.

Windows NT/2000/XP: The value of MB_SERVICE_NOTIFICATION changed starting with Windows NT 4.0. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of MessageBox. This mapping is only done for executables that have a version number less than 4.0, as set by the linker.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Microsoft Windows NT 3.x and Windows NT 4.0, you can do one of the following.

- At link-time, specify a version number less than 4.0
- At link-time, specify version 4.0. At run-time, use the GetVersionEx function to check the

system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

Windows 95/98/Me: Even though MessageBoxW exists, it is supported by the Microsoft Layer for Unicode to give more consistent behavior across all Windows operating systems. To use this, you must add certain files to your application, as outlined in .

Example

For an example, see Displaying a Message Box.

Function Information

Minimum DLL Version	user32.dll
Header	Declared in winuser.h, include windows.h
Import library	user32.lib
Minimum operating systems	Windows 95, Windows NT 3.1
Unicode	Implemented as ANSI and Unicode versions.

See Also

Dialog Boxes Overview, FlashWindow, MessageBeep, MessageBoxEx, MessageBoxIndirect, SetForegroundWindow

---

Platform SDK: Windows System Information

GetLocalTime

The GetLocalTime function retrieves the current local date and time.

```
void GetLocalTime(
  LPSYSTEMTIME lpSystemTime
);
```

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.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**

See Also

Local Time, **Time Functions**, **GetSystemTime**, **SetLocalTime**, **SYSTEMTIME**

Platform SDK: Windows System Information

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.

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. Additional information is provided by the IADsADSystemInfo interface.

The behavior of this function can be affected if the local computer is a node in a cluster. For more information, see ResUtilGetEnvironmentWithNetName and UseNetworkName.

```
BOOL GetComputerName(
  LPTSTR lpBuffer,
  LPDWORD lpnSize
);
```

Parameters

lpBuffer

[out] Pointer to a buffer that receives a null-terminated string containing the computer name or the cluster virtual server name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

lpnSize

[in, out] On input, specifies the size of the buffer, in TCHARs. On output, 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. The *lpnSize* parameter specifies the size of the buffer required, not including the terminating null character.

> **Windows Me/98/95: 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.

Example Code

For an example, see Getting System Information.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**
Unicode	**Implemented as GetComputerNameW (Unicode) and GetComputerNameA (ANSI). Note that Unicode support on Windows Me/98/95 requires Microsoft Layer for Unicode.**

See Also

Computer Names, System Information Functions, GetComputerNameEx, SetComputerName, SetComputerNameEx

---

Platform SDK: Windows System Information

GetUserName

The GetUserName function retrieves the name of the user associated with the current thread.

Use the GetUserNameEx function to retrieve the user name in a specified format. Additional information is provided by the IADsADSystemInfo interface.

```
BOOL GetUserName(
  LPTSTR lpBuffer,
  LPDWORD nSize
);
```

Parameters

lpBuffer

[out] Pointer to the buffer to receive the null-terminated string containing the user's logon name. If this buffer is not large enough to contain the entire user name, the function fails. A buffer size of (UNLEN + 1) characters will hold the maximum length user name including the terminating null character. UNLEN is defined in Lmcons.h.

nSize

[in, out] On input, this variable specifies the size of the *lpBuffer* buffer, in TCHARs. On output, the variable receives the number of TCHARs copied to the buffer, including the terminating null character.

If *lpBuffer* is too small, the function fails and GetLastError returns ERROR_INSUFFICIENT_BUFFER. This parameter receives the required buffer size, including the terminating null character.

If this parameter is greater than 32767, the function fails and GetLastError returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is a nonzero value, and the variable pointed to by *nSize* contains the number of TCHARs copied to the buffer specified by *lpBuffer*, including the terminating null character.

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

Remarks

If the current thread is impersonating another client, the GetUserName function returns the user name of the client that the thread is impersonating.

Example Code

For an example, see Getting System Information.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Advapi32.lib.**
DLL	**Requires Advapi32.dll.**
Unicode	**Implemented as GetUserNameW (Unicode) and GetUserNameA (ANSI). Note that Unicode support on Windows Me/98/95 requires Microsoft Layer for Unicode.**

See Also

System Information Functions, GetUserNameEx, LookupAccountName

---

Platform SDK: Memory Management

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.

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

Parameters

lpAddress

[in] The starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the nearest multiple of the allocation granularity. 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 and the allocation granularity on the host computer, use the GetSystemInfo function. If this parameter is NULL, the system determines where to allocate the region.

dwSize

[in] The size of the region, in bytes. 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] Type of memory allocation. This parameter must contain one 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 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 determining the current commitment state of each page.
MEM_RESERVE	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 VirtualAlloc function.
MEM_RESET	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. This value cannot be used with any other value. 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 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.

Windows Me/98/95: This flag is not supported.

This parameter can also specify the following values as indicated.

Value	Meaning
MEM_LARGE_PAGES	Allocates memory using large page support. The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.
MEM_PHYSICAL	Allocates physical memory with read-write access. This value is solely for use with Address Windowing Extensions (AWE) memory. This value must be used with MEM_RESERVE and no other values.
MEM_TOP_DOWN	Allocates memory at the highest possible address. **Windows Me/98/95:** This flag is not supported.
MEM_WRITE_WATCH	Causes the system to track pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE. 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. The write-tracking feature remains enabled for the memory region until the region is freed.

flProtect
[in] Memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the memory protection options, along with PAGE_GUARD or PAGE_NOCACHE as needed.

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

Each page has an associated page state. VirtualAlloc can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAlloc cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

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.

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, as mentioned previously.

To execute dynamically generated code, use VirtualAlloc to allocate memory and the VirtualProtect function to grant PAGE_EXECUTE access.

The VirtualAlloc function can be used to reserve an Address Windowing Extensions (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 decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

Example Code

For an example, see Reserving and Committing Memory.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**
Library	**Link to Kernel32.lib.**
DLL	**Requires Kernel32.dll.**

See Also

Virtual Memory Functions, Memory Management Functions, VirtualAllocEx, VirtualFree, VirtualLock, VirtualProtect, VirtualQuery

---

Platform SDK: Memory Management

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.

```
BOOL VirtualFree(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD dwFreeType
);
```

Parameters

lpAddress

[in] A pointer to the base address of the region of pages to be freed.

If the *dwFreeType* parameter is MEM_RELEASE, this parameter must be the base address returned by the VirtualAlloc function when the region of pages is reserved.

dwSize

[in] The size of the region of memory to be freed, in bytes.

If the *dwFreeType* parameter is MEM_RELEASE, this parameter must be 0 (zero). The function frees the entire region that is reserved in the initial allocation call to VirtualAlloc.

If the *dwFreeType* parameter is MEM_DECOMMIT, 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. If *lpAddress* is the base address returned by VirtualAlloc and *dwSize* is 0 (zero), the function decommits the entire region that is allocated by VirtualAlloc. After that, the entire region is in the reserved state.

dwFreeType

[in] The type of free operation. This parameter can be one of the following values.

Value	Meaning
MEM_DECOMMIT 0x4000	Decommits the specified region of committed pages. After the operation, the pages are in 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 the current commitment state. Do not use this value with MEM_RELEASE.
MEM_RELEASE 0x8000	Releases the specified region of pages. After this operation, the pages are in the free state. If you specify this value, *dwSize* must be 0 (zero), and *lpAddress* must point to the base address returned by the VirtualAlloc function when the region is reserved. The function fails if even of the conditions is not met. If any pages in the region are committed currently, 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 the current commitment state. Do not use this value with MEM_DECOMMIT.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call GetLastError.

Remarks

Each page of memory in a process virtual address space has a Page State. The VirtualFree 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. Subsequently, you can call VirtualAlloc to commit it, or VirtualFree to release it. Attempts to read from or write to a reserved page results in an access violation exception.

The VirtualFree 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 VirtualAlloc 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. After memory is released or decommited, 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 need to keep information, do not decommit or free memory that contains the information.

The VirtualFree function can be used on an AWE region of memory, and it invalidates any physical page mappings in the region when freeing the address space. However, the physical page is not deleted, and the application can use them. The application must explicitly call FreeUserPhysicalPages to free the physical pages. When the process is terminated, all resources are cleaned up automatically.

Example Code

For an example, see Reserving and Committing Memory.

Requirements

Client	Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server	Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header	Declared in Winbase.h; include Windows.h.
Library	Link to Kernel32.lib.
DLL	Requires Kernel32.dll.

See Also

Memory Management Functions, VirtualFreeEx, Virtual Memory Functions

---

Ayuda sobre tipos de estructuras definidas en los ficheros de cabecera del SDK

Platform SDK: DLLs, Processes, and Threads

PROCESS_INFORMATION

The PROCESS_INFORMATION structure is used with the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, or CreateProcessWithTokenW function. This structure contains information about the newly created process and its primary thread.

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```

Members

hProcess
Handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread
Handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId
Value that can be used to identify a process. The value is valid from the time the process is created until the time the process is terminated.

dwThreadId
Value that can be used to identify a thread. The value is valid from the time the thread is created until the time the thread is terminated.

Remarks

If the function succeeds, be sure to call the CloseHandle function to close the hProcess and hThread handles when you are finished with them. Otherwise, when the child process exits, the system cannot clean up these handles because the parent process did not close them. However, the system will close these handles when the parent process terminates, so they would be cleaned up at this point.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**

See Also

CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW

Platform SDK: DLLs, Processes, and Threads

STARTUPINFO

The STARTUPINFO structure is used with the CreateProcess, CreateProcessAsUser, and CreateProcessWithLogonW functions to specify the window station, desktop, standard handles, and appearance of the main window for the new process.

```
typedef struct _STARTUPINFO {
  DWORD cb;
  LPTSTR lpReserved;
  LPTSTR lpDesktop;
  LPTSTR lpTitle;
  DWORD dwX;
  DWORD dwY;
  DWORD dwXSize;
  DWORD dwYSize;
  DWORD dwXCountChars;
  DWORD dwYCountChars;
  DWORD dwFillAttribute;
  DWORD dwFlags;
  WORD wShowWindow;
  WORD cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFO,
*LPSTARTUPINFO;
```

Members

cb
Size of the structure, in bytes.

lpReserved
Reserved. Set this member to NULL before passing the structure to CreateProcess.

lpDesktop
Pointer to a null-terminated string that specifies either the name of the desktop, or the name of both the desktop and window station for this process. A backslash in the string indicates that the string includes both the desktop and window station names.

Windows Me/98/95: Desktops and window stations are not supported.

lpTitle
For console processes, this is the title displayed in the title bar if a new console window is created. If NULL, the name of the executable file is used as the window title instead. This parameter must be NULL for GUI or console processes that do not create a new console window.

dwX
If dwFlags specifies STARTF_USEPOSITION, this member is the x offset of the upper left corner of a window if a new window is created, in pixels. Otherwise, this member is ignored.

The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls CreateWindow to create an overlapped window if the *x* parameter of CreateWindow is CW_USEDEFAULT.

dwY
If dwFlags specifies STARTF_USEPOSITION, this member is the y offset of the upper left corner of a window if a new window is created, in pixels. Otherwise, this member is ignored.

The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls CreateWindow to create an overlapped window if the *y* parameter of CreateWindow is CW_USEDEFAULT.

dwXSize
If dwFlags specifies STARTF_USESIZE, this member is the width of the window if a new window is created, in pixels. Otherwise, this member is ignored.

For GUI processes, this is used only the first time the new process calls CreateWindow to create an overlapped window if the *nWidth* parameter of CreateWindow is CW_USEDEFAULT.

dwYSize
If dwFlags specifies STARTF_USESIZE, this member is the height of the window if a new window is created, in pixels. Otherwise, this member is ignored.

For GUI processes, this is used only the first time the new process calls CreateWindow to create an overlapped window if the *nHeight* parameter of CreateWindow is CW_USEDEFAULT.

dwXCountChars
If dwFlags specifies STARTF_USECOUNTCHARS, if a new console window is created in a console process, this member specifies the screen buffer width, in character columns. Otherwise, this member is ignored.

dwYCountChars
If dwFlags specifies STARTF_USECOUNTCHARS, if a new console window is created in a console process, this member specifies the screen buffer height, in character rows. Otherwise, this member is ignored.

dwFillAttribute
If dwFlags specifies STARTF_USEFILLATTRIBUTE, this member is the initial text and background colors if a new console window is created in a console application. Otherwise, this member is ignored.

This value can be any 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
```

dwFlags
Bit field that determines whether certain STARTUPINFO members are used when the process creates a window. This member can be one or more of the following values.

Value	Meaning
STARTF_FORCEONFEEDBACK	Indicates that the cursor is in feedback mode for two seconds after CreateProcess is called. The Working in Background cursor is displayed (see the Pointers tab in the Mouse control panel utility). If during those two seconds the process makes the first GUI call, the system gives five more seconds to the process. If during those five seconds the process shows a window, the system gives five more seconds to the process to finish drawing the window. The system turns the feedback cursor off after the first call to GetMessage, regardless of whether the process is drawing.
STARTF_FORCEOFFFEEDBACK	Indicates that the feedback cursor is forced off while the process is starting. The Normal Select cursor is displayed.
STARTF_RUNFULLSCREEN	Indicates that the process should be run in full-screen mode, rather than in windowed mode. This flag is only valid for console applications running on an x86 computer. Windows Me/98/95: This value is not supported.
STARTF_USECOUNTCHARS	If this value is not specified, the dwXCountChars and dwYCountChars members are ignored. Windows Me/98/95: This value is not supported.
STARTF_USEFILLATTRIBUTE	If this value is not specified, the dwFillAttribute member is ignored.
STARTF_USEPOSITION	If this value is not specified, the dwX and dwY members are ignored.
STARTF_USESHOWWINDOW	If this value is not specified, the wShowWindow member is ignored.
STARTF_USESIZE	If this value is not specified, the dwXSize and dwYSize members are ignored.
STARTF_USESTDHANDLES	Sets the standard input, standard output, and standard error handles for the process to the handles specified in the hStdInput, hStdOutput, and hStdError members of the STARTUPINFO structure. For this to work properly, the handles must be inheritable and the CreateProcess function's *bInheritHandles* parameter must be set to TRUE. For more information, see Handle

Inheritance.

If this value is not specified, the hStdInput, hStdOutput, and hStdError members of the STARTUPINFO structure are ignored.

wShowWindow
If dwFlags specifies STARTF_USESHOWWINDOW, this member can be any of the SW_ constants defined in Winuser.h. Otherwise, this member is ignored.

For GUI processes, wShowWindow specifies the default value the first time ShowWindow is called. The *nCmdShow* parameter of ShowWindow is ignored. In subsequent calls to ShowWindow, the wShowWindow member is used if the *nCmdShow* parameter of ShowWindow is set to SW_SHOWDEFAULT.

cbReserved2
Reserved for use by the C Run-time; must be zero.

lpReserved2
Reserved for use by the C Run-time; must be NULL.

hStdInput
If dwFlags specifies STARTF_USESTDHANDLES, this member is a handle to be used as the standard input handle for the process. Otherwise, this member is ignored.

hStdOutput
If dwFlags specifies STARTF_USESTDHANDLES, this member is a handle to be used as the standard output handle for the process. Otherwise, this member is ignored.

hStdError
If dwFlags specifies STARTF_USESTDHANDLES, this member is a handle to be used as the standard error handle for the process. Otherwise, this member is ignored.

Remarks

For graphical user interface (GUI) processes, this information affects the first window created by the CreateWindow function and shown by the ShowWindow function. For console processes, this information affects the console window if a new console is created for the process. A process can use the GetStartupInfo function to retrieve the STARTUPINFO structure specified when the process was created.

If a GUI process is being started and neither STARTF_FORCEONFEEDBACK or STARTF_FORCEOFFFEEDBACK is specified, the process feedback cursor is used. A GUI process is one whose subsystem is specified as "windows."

Requirements

Client	Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server	Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header	Declared in Winbase.h; include Windows.h.
Unicode	Implemented as STARTUPINFOW (Unicode) and STARTUPINFOA (ANSI).

See Also

CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, GetStartupInfo

Platform SDK: Windows System Information

SYSTEMTIME

The SYSTEMTIME structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME,
*PSYSTEMTIME;
```

Members

wYear
The year (1601 - 30827).

wMonth
The month.

January = 1
February = 2
March = 3
April = 4
May = 5
June = 6
July = 7
August = 8
September = 9
October = 10
November = 11
December = 12

wDayOfWeek
The day of the week.

Sunday = 0
Monday = 1
Tuesday = 2
Wednesday = 3
Thursday = 4
Friday = 5
Saturday = 6

wDay
The day of the month (1-31).

wHour
The hour (0-23).

wMinute
The minute (0-59).

wSecond
The second (0-59).

wMilliseconds
The millisecond (0-999).

Remarks

It is not recommended that you add and subtract values from the SYSTEMTIME structure to obtain relative times. Instead, you should

- **Convert the SYSTEMTIME structure to a FILETIME structure.**
- **Copy the resulting FILETIME structure to a ULARGE_INTEGER structure.**
- **Use normal 64-bit arithmetic on the ULARGE_INTEGER value.**

The system can periodically refresh the time by synchronizing with a time source. Because the system time can be adjusted either forward or backward, do not compare system time readings to determine elapsed

time. Instead, use one of the methods described in Windows Time.

Requirements

Client	**Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.**
Server	**Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.**
Header	**Declared in Winbase.h; include Windows.h.**

See Also

FILETIME, **FileTimeToSystemTime**, **GetSystemTime**, **ULARGE_INTEGER**, **SetSystemTime**
