



# TaskGhost Technical Reference

Version 1.1

Monday, April 16, 2007

This software is provided as-is.  
There are no warranties, expressed or implied.

Copyright© 2006 Tomasello Software, LLC.  
All rights reserved

**TaskGhost® is a Registered Trademark of Tomasello Software  
LLC.**

## Table of Contents

Table of Contents.....	1
TaskGhost Functions.....	2
General purpose.....	2
Print.....	2
CheckTime.....	3
DestroyCronContext.....	4
Run.....	5
FreeHandle.....	7
SendKeys.....	7
SendMail.....	11
Local Process Creation.....	12
CreateProcess.....	12
CreateProcessAsUser.....	14
ShellExecute.....	18
User Impersonation.....	20
LogonUser.....	20
ImpersonateLoggedOnUser.....	24
RevertToSelf.....	25
Network Process Creation.....	25
NetCreateProcessAsUser.....	25

NetCreateProcess .....	29
NetScheduleJobAddAsUser .....	32
NetScheduleJobAdd .....	35
NetScheduleJobDel .....	37
NetScheduleJobGetInfo.....	38
NetScheduleJobGetInfoTime.....	39
NetScheduleJobGetInfoError.....	40
NetScheduleJobGetInfoCommand .....	41
NetScheduleJobGetInfoJobld .....	42
NetScheduleJobEnum .....	42
NetScheduleJobEnumCount .....	43
NetScheduleJobEnumGet .....	44
Windows and Messaging .....	45
DispatchMessage .....	45
GetMessage .....	45
GetMessageId .....	47
GetMessageWParam .....	47
GetMessageLParam.....	48
KillTimer.....	48
PumpMessage.....	49
SetTimer .....	50
FindWindow .....	51
SetForegroundWindow .....	52
Environment Variables and Encryption .....	53
GetEnvironmentVariable.....	53
SetEnvironmentVariable .....	55
File Transfer Protocol – FTP .....	56
FtpCd.....	56
FtpConnect .....	57
FtpDel .....	59
FtpGet.....	60
FtpMkDir .....	61
FtpPut.....	62
FtpRmDir .....	63
FtpRename.....	64
FtpTest .....	65

## TaskGhost Functions

### General purpose

#### Print

#### Print

Prints a string to the TaskGhost console window.

## Void Print string

### Arguments:

*string*

The string to print.

### Remarks:

The Print function is useful both as a debugging aid and a way to convey status information to the TaskGhost user.

The generic scripts that ship with TaskGhost use Print liberally to convey status and errors to the user.

It should be noted however that excessive output, tight loops that output many thousands of messages, can fill the TaskGhost IO queue. When this happens, TaskGhost may purposely ignore some messages in order to allow TaskGhost to catch up.

**In the following code we use Print to output "Hello World" 1024 times.**

```
Sub main(args)

    Dim jx
    For jx = 1 To 1024
        TGCtrl.Print (jx & " Hello World")
    Next

End Sub
```

### Requirements:

Version 1.0

### See Also:

NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## CheckTime

### CheckTime

This function tests the current date/time to see if it matches a cron specification.

**Bool** CheckTime (id, Reset, CronSpec)

### Arguments:

*id*

An integer that uniquely identifies the cron specification.

*Reset*

When True, the internal cron state for the cron specification identified by identifier id is reset. See the remarks section for more information on the Reset flag.

### *CronSpec*

The CronSpec is a UNIX style cron specification used to match the current date/time. Please see the section on extended cron syntax for a complete description of cron syntax.

#### **Returns:**

This function returns **True** if the current date/time match the CronSpec, **False** otherwise.

#### **Remarks:**

The CheckTime function is the primary mechanism for job scheduling within TaskGhost. The CheckTime function is used to test the current time to see if it matches the specified time.

CheckTime uses an extended version of UNIX 'cron' syntax to specify complex date/time matching patterns.

Please see the section on extended cron syntax for a full description of using cron syntax for scheduling.

The Reset flag is typically used to reset the internal state of a cron object in response to either a manual or automatic time change, i.e., Daylight Savings.

The user will not normally need to worry about resetting the cron object as this is handled automatically from within the main scheduling loop of the TaskGhost scheduler.

**The following code snippet from Schedule.vbs uses CheckTime to schedule Hello.vbs to run each Mon-Fri at 4:30 AM.**

```
" Run the "hello world" script Mon-Fri at 0430 (04:30AM).  
If TGCtrl.CheckTime (7, tc, "30 4 * *") = True Then  
    TGCtrl.Run RUN_ASYNC, "hello.vbs", vbNullString  
End If
```

#### **Requirements:**

Version 1.0

#### **See Also:**

[DestroyCronContext](#), [Run](#)

Schedule.vbs in the TaskGhost\Scripts directory.

## **DestroyCronContext**

### **DestroyCronContext**

The DestroyCronContext function deletes a range of cron-context structures from the environment of the current thread (script.)

## **Long DestroyCronContext (MinCronId, MaxCronId)**

### **Arguments:**

#### *MinCronId*

Specifies a minimum cron identifier. Cron-context structures with a cron identifier smaller than *MinCronId* will not be deleted.

#### *MaxCronId*

Specifies a maximum cron identifier. Cron-context structures with a cron identifier larger than *MaxCronId* will not be deleted.

### **Returns:**

Returns the number of cron-context structures destroyed.

### **Remarks:**

All cron triggers allocated via calls to [CheckTime](#) store context information in the threads environment. This context information keeps track of the last time-period in which a cron pattern was satisfied - triggered.

If this context information is not cleared when the script creating the trigger expires, then new jobs starting up in the same thread that use the same cron identifier will assume the context information of the last cron trigger.

This is not necessarily a bad thing, but script writers should be aware of this pseudo-persistent context information.

The main loop of Schedule.vbs calls this function before exiting to insure that triggers allocated in Schedule.vbs, die with Schedule.vbs.

The DestroyCronContext function deletes all jobs whose job identifiers are in the range *MinCronId* through *MaxCronId*.

### **Requirements:**

Version 1.0

### **See Also:**

[NetScheduleJobEnum](#), [NetScheduleJobAdd](#)  
Schedule.vbs in the TaskGhost\Scripts directory.

## **Run**

### **Run**

Run a VBScript source module by calling the main function and passing it the CommandLine.

## **Variant Run (Flags, Filename, CommandLine)**

### **Arguments:**

#### *Flags*

Specifies the way in which to run the script. This value can be zero in

which case the script will be run with all default values. I.e., synchronously.

This parameter can be one or more of the following values combined with the Or operator:

<b>RUN_A SYNC</b>	Runs the specified script asynchronously. This flag should not be used if the return value from the script run is desired.
-----------------------	---

*Filename*

The name of the script to run.

This script must contain a main function as described in the section titled script syntax.

A full path to the script can be given, or no path if the script is in the standard TaskGhost\Scripts directory.

*CommandLine*

This parameter is the command line argument passed to the main function of the named script. For more information on this command line, please see the Remarks section.

**Return Value:**

If the script is run synchronously, then the return value is the value returned from the called script's main function. If the script is run with the RUN\_ASYNC flag, then the function returns vbNullString.

**Remarks:**

When calling a script via the Run function, you are free to pass any command line you wish. However, when a script is loaded and run via the TaskGhost User Interface, TaskGhost passes the current date/time as the command line.

Because of this behavior, some scripts may not run correctly when run from the TaskGhost UI.

**The following code snippet from Schedule.vbs uses CheckTime to schedule Hello.vbs to run each Mon-Fri at 4:30 AM.**

```
" Run the "hello world" script Mon-Fri at 0430 (04:30AM).  
If TGCtrl.CheckTime (7, tc, "30 4 * *") = True Then  
    TGCtrl.Run RUN_ASYNC, "hello.vbs", vbNullString  
End If
```

**Requirements:**

Version 1.0

**See Also:**

## [CheckTime](#)

Schedule.vbs in the TaskGhost\Scripts directory.

## **FreeHandle**

### **FreeHandle**

The FreeHandle function the specified object handle.

### **Bool FreeHandle (hObject)**

#### **Arguments:**

*hObject*

Specifies the object handle to free.

#### **Returns:**

If the function succeeds, the return value is **True**.

If the function fails, the return value is **False**.

#### **Remarks:**

The FreeHandle function closes and frees any and all object handle types returned from the TaskGhost API.

The following functions return these handles.

[LogonUser](#)

[GetMessage](#)

[NetScheduleJobGetInfo](#)

[NetScheduleJobEnumGet](#)

#### **Requirements:**

Version 1.0

#### **See Also:**

Schedule.vbs in the TaskGhost\Scripts directory.

NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **SendKeys**

### **SendKeys**

Sends one or more keystrokes to the active window (as if typed on the keyboard).

### **Void SendKeys string**

#### **Arguments:**

*string*

String value indicating the keystroke(s) you want to send.

**Remarks:**

Use the **SendKeys** method to send keystrokes to applications that have no automation interface. Most keyboard characters are represented by a single keystroke. Some keyboard characters are made up of combinations of keystrokes (CTRL+SHIFT+HOME, for example). To send a single keyboard character, send the character itself as the *string* argument. For example, to send the letter x, send the *string* argument "x".

Note To send a space, send the string " ".

You can use **SendKeys** to send more than one keystroke at a time. To do this, create a compound string argument that represents a sequence of keystrokes by appending each keystroke in the sequence to the one before it. For example, to send the keystrokes a, b, and c, you would send the string argument "abc". The **SendKeys** method uses some characters as modifiers of characters (instead of using their face-values). This set of special characters consists of parentheses, brackets, braces, and the:

- ✘ plus sign "+",
- ✘ caret "^",
- ✘ percent sign "%",
- ✘ and tilde "~"

Send these characters by enclosing them within braces "{}". For example, to send the plus sign, send the string argument "{+}". Brackets "[ ]" have no special meaning when used with **SendKeys**, but you must enclose them within braces to accommodate applications that do give them a special meaning (for dynamic data exchange (DDE) for example).

- ✘ To send bracket characters, send the string argument "{}[]" for the left bracket and "{}]" for the right one.
- ✘ To send brace characters, send the string argument "{}{}" for the left brace and "{}}" for the right one.

Some keystrokes do not generate characters (such as ENTER and TAB). Some keystrokes represent actions (such as BACKSPACE and BREAK). To send these kinds of keystrokes, send the arguments shown in the following table:

Key	Argument
BACKSPACE	{BACKSPACE}, {BS}, or {BKSP}

BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}
DEL or DELETE	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER	{ENTER} or ~
ESC	{ESC}
HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}

F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

To send keyboard characters that are comprised of a regular keystroke in combination with a SHIFT, CTRL, or ALT, create a compound string argument that represents the keystroke combination. You do this by preceding the regular keystroke with one or more of the following special characters:

Key	Special Character
SHIFT	+
CTRL	^
ALT	%

**Note** When used this way, these special characters are not enclosed within a set of braces.

To specify that a combination of SHIFT, CTRL, and ALT should be held down while several other keys are pressed, create a compound string argument with the modified keystrokes enclosed in parentheses. For example, to send the keystroke combination that specifies that the SHIFT key is held down while:

- ✘ e and c are pressed, send the string argument "+(ec)".
- ✘ e is pressed, followed by a lone c (with no SHIFT), send the string argument "+ec".

You can use the **SendKeys** method to send a pattern of keystrokes that consists of a single keystroke pressed several times in a row. To do this, create a compound string argument that specifies the keystroke you want to repeat, followed by the number of times you want it repeated. You do this using a compound string argument of the form *{keystroke number}*. For example, to send

the letter "x" ten times, you would send the string argument "{x 10}". Be sure to include a space between keystroke and number.

**Note** The only keystroke pattern you can send is the kind that is comprised of a single keystroke pressed several times. For example, you can send "x" ten times, but you cannot do the same for "Ctrl+x".

**Note** You cannot send the PRINT SCREEN key {PRTSC} to an application.

### Requirements:

Version 1.1

### See Also:

[FindWindow](#), [SetForegroundWindow](#)

SendKeys.vbs in the TaskGhost\Sample Scripts directory.

## SendMail

### SendMail

Send mail using the SMTP service.

**Void** SendMail from, to, subject, body, attachment

### Arguments:

*from*

The *from* email address.

*to*

The *to* email address.

*subject*

The *subject* text.

*body*

The *body* text.

*attachment*

Optional attachment.

### Returns:

none.

### Remarks:

Internet information service (IIS) installs an SMTP service on the local computer. IIS ships with Windows XP, 2000, and 2003. The SMTP service must be running in order for email to function properly.

## Requirements:

Version 1.1

## See Also:

[email](#),

EEmail.vbs in the TaskGhost\Sample Scripts directory.

## Local Process Creation

### CreateProcess

#### 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](#).

#### Long CreateProcess (flags, CommandLine)

##### Arguments:

*flags*

The flags parameter modify the default behavior of CreateProcess.

The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the [Or](#) operator.

<b>CP_HID E</b>	Hides the window associated with the application.
<b>CP_ASYNC</b>	Runs the process asynchronously.
<b>CP_WAIT_IDLE</b>	The <b>CP_WAIT_IDLE</b> flag causes the script to wait until the specified process is waiting for user input with no input pending, or until and internal 15 second timeout occurs. Note: Must be be used with <b>CP_ASYNC</b> .

*CommandLine*

String that specifies the command line to execute. The maximum length of this string is 32K characters.

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. If the file name does not contain an extension, each of .com, .exe, .cmd, and .bat are tried in order. 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.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

#### Returns:

The return value from a synchronous CreateProcess (CP\_ASYNC not specified in *flags*) is the exit status of the new process. The return value from an asynchronous CreateProcess (CP\_ASYNC specified in *flags*) is zero if the process was successfully spawned.

The exit status is 0 if the process terminated normally. A spawned process can set the exit status to a nonzero value if the spawned process specifically calls the exit routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started).

#### Remarks:

When CreateProcess is called with the CP\_ASYNC flag, CreateProcess does not wait for the new process to finish initialization before returning. This lag between the time a process is created and the time at which the new process is fully initialized can cause some problems if you are trying to initiate communication with the new process too soon.

If your script must communicate or otherwise interact with a freshly the spawned process, it is recommended you use the CP\_WAIT\_IDLE flag to insure the process has entered a quiescent state.

**This script (CreateProcess.vbs) shows how to call CreateProcess and how to capture the return code.**

```

Sub main(args)
    Dim commandline
    commandline = args
    Dim exit_code
    exit_code = TGCtrl.CreateProcess(0,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub

```

**Requirements:**

Version 1.0

**See Also:**

[CreateProcessAsUser](#), [ImpersonateLoggedOnUser](#), [LogonUser](#)  
CreateProcess.vbs and CreateProcessAsUser.vbs in the TaskGhost\Scripts directory.

**CreateProcessAsUser**

**CreateProcessAsUser**

The CreateProcessAsUser function creates a new process and its primary thread. The new process then runs the specified executable file. The CreateProcessAsUser function is similar to the [CreateProcess](#) function, except that the new process runs in the security context of the user represented by the *user* parameter. By default, the new process is non-interactive, that is, it runs on a desktop that is not visible and cannot receive user input. Also, by default, the new process inherits the environment of the calling process, rather than the environment associated with the specified user.

**Long CreateProcessAsUser (flags, user, domain, password, CommandLine)**

**Arguments:**

*flags*

The flags parameter modify the default behavior of CreateProcessAsUser. The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the [Or](#) operator.

<b>CP_I NTER ACTI VE</b>	<p>Specifies whether the application (process) provides a user interface on a desktop that can be used by whoever is logged on when the application is started.</p> <p>When running as a service, this option is available only if the service is</p>
--------------------------------------	---

	running as a LocalSystem account. Cannot be combined with CP_NTI.
<b>CP_HIDE</b>	Hides the window associated with the application.
<b>CP_ASYNC</b>	Runs the process asynchronously.
<b>CP_PROFILE</b>	Populates the HKEY_CURRENT_USER registry key with the specified user's profile information. This is consistent with a normal interactive logon.
<b>CP_WAIT_IDLE</b>	The CP_WAIT_IDLE flag causes the script to wait until the specified process is waiting for user input with no input pending, or until an internal 15 second timeout occurs. Note: Must be used with CP_ASYNC.

*user*

String that specifies the name of the user. This is the name of the user account to log on to. If you use the UPN format, `user@DNS_domain_name`, the *domain* parameter must be [vbNullString](#).

The user account must have Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

*domain*

String that specifies the name of the domain or server whose account database contains the *user* account. If this parameter is [vbNullString](#), the user name must be specified in UPN format. If this parameter is ".", the function validates the account using only the local account database.

*password*

String that specifies the clear-text password for the user account specified by *user*. For more information about protecting passwords, see [Handling Passwords](#).

*CommandLine*

String that specifies the command line to execute. The maximum length of this string is 32K characters.

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. If the file name does not contain an extension, each of .com, .exe, .cmd, and .bat are tried in order. 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.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

#### Returns:

The return value from a synchronous `CreateProcessAsUser` (`CP_ASYNC` not specified in *flags*) is the exit status of the new process. The return value from an asynchronous `CreateProcessAsUser` (`CP_ASYNC` specified in *flags*) is zero if the process was successfully spawned.

The exit status is 0 if the process terminated normally. A spawned process can set the exit status to a nonzero value if the spawned process specifically calls the exit routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started).

#### Remarks:

When `CreateProcessAsUser` is called with the `CP_ASYNC` flag, `CreateProcessAsUser` does not wait for the new process to finish initialization before returning. This lag between the time a process is created and the time at which the new process is fully initialized can cause some problems if you are trying to initiate communication with the new process too soon.

If your script must communicate or otherwise interact with a freshly spawned process, it is recommended you use the `CP_WAIT_IDLE` flag to insure the process has entered a quiescent state.

By default, `CreateProcessAsUser` does not load the specified user's profile into the `HKEY_USERS` registry key. This means that access to information in the `HKEY_CURRENT_USER` registry key may not produce results consistent with a normal interactive logon. If the new process requires the specified user's profile, then you should pass `CP_PROFILE` in the *flags* parameter.

You should only load a user's profile when it is required as it can be a time consuming process.

If you receive the error "*A required privilege is not held by the client*" or "*Access Denied*" then your login account does not have sufficient privileges to allow your thread (the thread your script is running in) to impersonate the specified user. This can happen even if you are an Administrator on the local machine. This error usually goes away when you run TaskGhost as a system service because processes running in the LocalSystem account have elevated privileges.

Domain Administrators will not have this problem, nor will Administrators within a workgroup or standalone machine. However if you are an administrator on a particular machine which is part of a domain and you do not have domain administrative privileges, then you are likely to see this error.

To resolve this problem, you'll need to elevate the rights of the account calling CreateProcessAsUser with the "*Replace a process level token*" right. To do so, open the *Control Panel / Administrative Tools / Local Security Policy* and add the user account to the "*Replace a process level token*" right. (You may have to logout or even reboot to have this change take effect.)

**This script (CreateProcessAsUser.vbs) shows how to call CreateProcessAsUser and how to capture the return code.**

```
Sub main(args)
    Dim user
    Dim domain
    Dim password
    Dim commandline
    " set these in your environment, or here in the script
    " If you're running as a service, you will need to set these
    " as SYSTEM variables
    user = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT, "USER")
    domain = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT,
"DOMAIN")
    password = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT Or
TSE_ENV_CRYPT, "PASSWORD")
    commandline = args
    Dim exit_code
    exit_code =
TGCtrl.CreateProcessAsUser(0,user,domain,password,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub
```

## Requirements:

Version 1.0

## See Also:

[CreateProcess](#), [ImpersonateLoggedOnUser](#), [LogonUser](#)

CreateProcessAsUser.vbs and CreateProcessAsUserInt.vbs in the TaskGhost\Scripts directory.

## ShellExecute

### ShellExecute

Performs an operation on a specified file.

### Long ShellExecute (flags, verb, file, params)

#### Arguments:

*flags*

The flags parameter modify the default behavior of ShellExecute.

The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the [Or](#) operator.

CP_HIDE	Hides the window associated with the application.
CP_ASYNC	Runs the process asynchronously.

*verb*

A string that specifies the action to be performed. The set of available verbs depends on the particular file or folder. Generally, the actions available from an object's shortcut menu are available verbs.

The following verbs are commonly used:

Verb	Meaning
"edit"	Launches an editor and opens the document for editing. If <i>file</i> is not a document file, the function will fail.
"explore"	Explores the folder specified by <i>file</i> .
"find"	Initiates a search starting from the specified directory.
"open"	Opens the file specified by the <i>file</i> parameter. The file can be an executable file, a document file, or a folder.

"print"	Prints the document file specified by <i>file</i> . If <i>file</i> is not a document file, the function will fail.
vbNullString	For Windows 2000 and later systems, the default verb is used if available. If not, the "open" verb is used. If neither verb is available, the system uses the first verb listed in the registry.

*file*

String that specifies the file or object on which to execute the specified verb. To specify a Shell namespace object, pass the fully qualified parse name. Note that not all verbs are supported on all objects. For example, not all document types support the "print" verb.

*params*

If the *file* parameter specifies an executable file, *params* is a string that specifies the parameters to be passed to the application. The format of this string is determined by the verb that is to be invoked. If *file* specifies a document file, *params* should be `vbNullString`.

#### Returns:

The return value from a synchronous ShellExecute (CP\_ASYNC not specified in *flags*) is the exit status of the new process. The return value from an asynchronous ShellExecute (CP\_ASYNC specified in *flags*) is zero if the process was successfully spawned.

The exit status is 0 if the process terminated normally. A spawned process can set the exit status to a nonzero value if the spawned process specifically calls the exit routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started).

#### Remarks:

When ShellExecute is called with the CP\_ASYNC flag, ShellExecute does not wait for the new process to finish initialization. This lag between the time a process is created and the time at which the new process is fully initialized can cause some problems if you are trying to initiate communication with the new process too soon.

#### Examples:

To open a folder, use either of the following calls:

```
ShellExecute(0, vbNullString, <fully_qualified_path_to_folder>,
vbNullString, vbNullString);
```

or

```
ShellExecute(0, "open", <fully_qualified_path_to_folder>, vbNullString,  
vbNullString);
```

To explore a folder, use:

```
ShellExecute(0, "explore", <fully_qualified_path_to_folder>, vbNullString,  
vbNullString);
```

To launch the Shell's Find utility for a directory, use:

```
ShellExecute(0, "find", <fully_qualified_path_to_folder>, vbNullString,  
vbNullString);
```

If *verb* is `vbNullString`, the function opens the file specified by *file*. If *verb* is "open" or "explore", the function will attempt to open or explore the folder.

**This script (ShellExecute.vbs) shows how to launch a document using ShellExecute.**

```
Sub main(args)  
    Dim exit_code  
    exit_code = TGCtrl.ShellExecute(0,"open","c:\forcast.doc",vbNullString)  
    If exit_code = -1 Then  
        TGCtrl.Print("Process failed to start!")  
    Else  
        TGCtrl.Print("Process returned " & exit_code)  
    End If  
End Sub
```

### Requirements:

Version 1.0

### See Also:

[CreateProcessAsUser](#), [CreateProcess](#)  
ShellExecute.vbs in the TaskGhost\Scripts directory.

## User Impersonation

### LogonUser

### LogonUser

This function tests the current date/time to see if it matches a cron specification.

## Handle LogonUser (user, domain, password, LogonType, LogonProvider)

### Arguments:

*user*

String that specifies the name of the user. This is the name of the user account to log on to. If you use the UPN format, *user@DNS\_domain\_name*, the *domain* parameter must be [vbNullString](#).

The user account must have Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

*domain*

String that specifies the name of the domain or server whose account database contains the *user* account. If this parameter is [vbNullString](#), the user name must be specified in UPN format. If this parameter is ".", the function validates the account using only the local account database.

*password*

String that specifies the clear-text password for the user account specified by *user*. For more information about protecting passwords, see [Handling Passwords](#).

*logonType*

Specifies the type of logon operation to perform. This parameter can be one of the following values.

Value	Meaning
LOGON32_LOGON_BATCH	This logon type is intended for batch servers, where processes may be executing on behalf of a user without their direct intervention; or for higher performance servers that process many clear-text authentication attempts at a time, such as mail or web servers. The LogonUser function does not cache credentials for this logon type.
LOGON32_LOGON_INTERACTIVE	This logon type is intended for users who will be interactively using the computer, such as a user being logged on by a terminal server, remote shell, or similar process. This logon type has the additional expense of caching logon information for disconnected operation, and is therefore inappropriate for some client/server applications, such as a mail server.
LOGON32_LOGON_NETWORK	This logon type is intended for high performance servers to authenticate clear text passwords. The LogonUser function does not cache credentials for this logon type.
LOGON32_LOGON_NETWORK_	This logon type preserves the name and password in the authentication packages, allowing the server to make connections to other network servers while impersonating the client. This allows a server to accept

<b>CLEARTEXT</b>	clear text credentials from a client, call LogonUser, verify that the user can access the system across the network, and still communicate with other servers.
<b>LOGON32_LOGON_NEW_CREDENTIALS</b>	This logon type allows the caller to clone its current token and specify new credentials for outbound connections. The new logon session has the same local identity, but uses different credentials for other network connections. This logon type is supported only by the <b>LOGON32_PROVIDER_WINNT50</b> logon provider.
<b>LOGON32_LOGON_SERVICE</b>	Indicates a service-type logon. The account provided must have the service privilege enabled.
<b>LOGON32_LOGON_UNLOCK</b>	This logon type is intended for <i>GINA</i> DLLs logging on users who will be interactively using the computer. This logon type allows a unique audit record to be generated that shows when the workstation was unlocked.

*logonProvider*

Specifies the logon provider. This parameter can be one of the following values.

<b>Value</b>	<b>Meaning</b>
<b>LOGON32_PROVIDER_DEFAULT</b>	Use the standard logon provider for the system. The default security provider is NTLM.  The default provider is negotiate, unless you pass NULL for the domain name and the user name is not in UPN format. In this case the default provider is NTLM.
<b>LOGON32_PROVIDER_WINNT50</b>	Windows XP/2000: Use the negotiate logon provider.
<b>LOGON32_PROVIDER_WINNT40</b>	Use the NTLM logon provider.
<b>LOGON32_PROVIDER_WINNT35</b>	Use the Windows NT 3.5 logon provider.

**Returns:**

If the function succeeds, the return value is a handle represents the specified user.

If the function fails, the return value is zero.

You can use the returned handle in calls to the [ImpersonateLoggedOnUser](#) function.

When you no longer need this handle, close it by calling the [CloseHandle](#) function.

**Remarks:**

You can use LogonUser with [ImpersonateLoggedOnUser](#) to gain access to system resources using the access rights and security settings of a domain user. This access is not limited to simply launching a process, but resources such as disk files, registry keys, etc., may also be accessed.

Use the [FreeHandle](#) to close this handle and free any associated resources.

**The following sample script, Impersonate.vbs, uses LogonUser and ImpersonateLoggedOnUser to gain access to system resources which might otherwise be unavailable.**

```
Sub main(args)
Dim user
Dim domain
Dim password
" set these in your environment, or here in the script
" If you're running as a service, you will need to set these
" as SYSTEM variables
user = TGCtrl.GetEnvironmentVariable
(TSE_ENV_PERSISTENT, "USER")
domain = TGCtrl.GetEnvironmentVariable
(TSE_ENV_PERSISTENT, "DOMAIN")
password = TGCtrl.GetEnvironmentVariable
(TSE_ENV_PERSISTENT Or TSE_ENV_CRYPT, "PASSWORD")
" logon as user X
Dim hToken
hToken =
TGCtrl.LogonUser(user,domain,password,LOGON32_LOGON_BATCH,LOGO
N32_PROVIDER_DEFAULT)
" Impersonate user X
Dim ilu
ilu = TGCtrl.ImpersonateLoggedOnUser(hToken)
" do whatever you want to do as user X
If ilu = true Then
```

```
        TGCtrl.Print ("This thread is now running as " + user)
Else
    TGCtrl.Print ("Unable to become " + user)
End If
" all done with this user
TGCtrl.FreeHandle(hToken)
" terminate impersonation for this thread
TGCtrl.RevertToSelf
End Sub
```

### Requirements:

Version 1.0

### See Also:

[RevertToSelf](#), [ImpersonateLoggedOnUser](#)  
Impersonate.vbs in the TaskGhost\Scripts directory.

## ImpersonateLoggedOnUser

### ImpersonateLoggedOnUser

The ImpersonateLoggedOnUser function lets the calling thread impersonate the security context of a logged-on user. The user is represented by a token handle.

### Bool ImpersonateLoggedOnUser (hToken)

#### Arguments:

*hToken*

Handle to an access token that represents a logged-on user. This handle is returned by a call to [LogonUser](#).

#### Returns:

If the function succeeds, the return value is [True](#).

If the function fails, the return value is [False](#).

#### Remarks:

The impersonation lasts until the thread exits or until it calls [RevertToSelf](#).

The calling thread does not need to have any particular privileges to call ImpersonateLoggedOnUser.

If the call to ImpersonateLoggedOnUser fails, the client connection is not impersonated and the client request is made in the security context of the process.

All impersonate functions, including ImpersonateLoggedOnUser, check to determine if the caller has the SeImpersonatePrivilege privilege. If the caller has the SeImpersonatePrivilege privilege, or if the authenticated identity is the same

as the caller, then the requested impersonation is allowed. Otherwise, the impersonation succeeds at Identify level only.

**Please see the code sample for [LogonUser](#) for an example of how to use this function.**

**Requirements:**

Version 1.0

**See Also:**

[RevertToSelf](#), [LogonUser](#)

Impersonate.vbs in the TaskGhost\Scripts directory.

## **RevertToSelf**

### **RevertToSelf**

The RevertToSelf function terminates the impersonation of a client application.

### **Bool RevertToSelf ()**

**Returns:**

If the function succeeds, the return value is **True**.

If the function fails, the return value is **False**.

**Remarks:**

If RevertToSelf fails, your application continues to run in the context of the client, which is not appropriate. You should shut down the process if RevertToSelf fails.

**Please see the code sample for [LogonUser](#) for an example of how to use this function.**

**Requirements:**

Version 1.0

**See Also:**

[ImpersonateLoggedOnUser](#), [LogonUser](#)

Impersonate.vbs in the TaskGhost\Scripts directory.

## **Network Process Creation**

### **NetCreateProcessAsUser**

#### **NetCreateProcessAsUser**

The NetCreateProcessAsUser function creates a new process on a local or remote computer. The new process then runs the specified executable file.

The NetCreateProcessAsUser function is similar to the [NetCreateProcess](#) function, except that the new process runs in the security context of the

user represented by the *user* parameter. By default, the new process is non-interactive, that is, it runs on a desktop that is not visible and cannot receive user input.

The NetCreateProcessAsUser function submits a job to the Windows AT Service to run immediately on the specified computer. This function requires that the Windows schedule service be started at the computer to which the job is submitted.

The NetCreateProcessAsUser function sets the AT Service account name and password before creating the process. The AT Service account name and password are used as the credentials for scheduled jobs created with NetCreateProcessAsUser. NetCreateProcessAsUser impersonates the caller. The caller must be a member of the Administrators group for this function to succeed.

### **Long NetCreateProcessAsUser (flags, user, server, password, CommandLine)**

#### **Arguments:**

*flags*

The flags parameter modify the default behavior of NetCreateProcessAsUser.

The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the [Or](#) operator.

<b>NCP_I NTER ACTI VE</b>	Specifies whether the application (process) provides a user interface on a desktop that can be used by whoever is logged on when the application is started.
---------------------------------------	--

*user*

String that specifies the name of the user. This is the name of the user account to log on to.

The user account must have Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is "", the local computer is used.

This string must begin with \\.

*password*

String that specifies the clear-text password for the user account specified by *user*. For more information about protecting passwords, see [Handling Passwords](#).

### *CommandLine*

String that specifies the command line to execute. The maximum length of this string is 32K characters.

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. 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:

- ✘ The directory from which the application loaded.
- ✘ The current directory for the parent process.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

Note: If a fully qualified PATH to the program to execute on the remote computer is not given, then you must insure that the PATH variable on the remote computer is setup correctly (to find the program).

Because the "System Environment Variables" will show-through for all users, it is recommended that the "System Environment Variables (PATH)" be configured correctly on the remote computer.

On Windows 2000, you can setup the "System Environment Variables" from the *Control Panel/System/Advanced/Environment Variables/System Variables*.

### **Returns:**

A positive integer if successful.

A return value of -1 indicates an error (the new process is not started).

If the process is successfully created, the return value is a positive integer that represents the AT Service "Job ID". This Job Id is only valid while the job is running.

### **Remarks:**

The NetCreateProcessAsUser function creates a one-time AT Service job that executes immediately.

The primary benefit of using NetCreateProcessAsUser over [NetScheduleJobAddAsUser](#) is that you can use the rich scheduling syntax of TaskGhost to invoke the call instead of being restricted to the limited AT Service scheduling capabilities.

Things one might consider when deciding whether to use NetCreateProcessAsUser or NetScheduleJobAddAsUser might be:

- ✘ Will TaskGhost be running during those times when the job needs to run, and therefore be available to schedule it?
- ✘ Does the schedule require the rich scheduling capabilities of TaskGhost?

If you answered YES to both of the above, then NetCreateProcessAsUser is a better choice.

Only members of the Administrators local group can successfully execute the NetCreateProcessAsUser function to schedule a job on a remote server.

**This script (NetCreateProcessAsUser.vbs) shows how to call NetCreateProcessAsUser and how to capture the return code.**

```
Sub main(args)
    Dim user
    Dim server
    Dim password
    Dim commandline
    " set these in your environment, or here in the script
    " If you're running as a service, you will need to set these
    " as SYSTEM variables
    user = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT, "USER")
    domain = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT,
"SERVER")
    password = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT Or
TSE_ENV_CRYPT, "PASSWORD")
    commandline = args
    Dim exit_code
    exit_code =
TGCtrl.NetCreateProcessAsUser(0,user,domain,password,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub
```

**Requirements:**

### Version 1.1

The *user* and *password* parameters are currently ignored and won't be available until Version 1.1.

#### See Also:

[NetCreateProcess](#), [CreateProcess](#), [NetScheduleJobAdd](#),  
[NetScheduleJobAddAsUser](#),

NetCreateProcessAsUser.vbs in the TaskGhost\Scripts directory.

## NetCreateProcess

### NetCreateProcess

The NetCreateProcess function creates a new process on a local or remote computer. The new process then runs the specified executable file. By default, the new process is non-interactive, that is, it runs on a desktop that is not visible and cannot receive user input. Also, by default, the new process inherits the environment of the calling process, rather than the environment associated with the specified user.

The NetCreateProcess function submits a job to the Windows AT Service to run immediately on the specified computer. This function requires that the Windows schedule service be started at the computer to which the job is submitted.

### Long NetCreateProcess (flags, server, CommandLine)

#### Arguments:

*flags*

The flags parameter modify the default behavior of NetCreateProcess.

The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the [Or](#) operator.

<b>NCP_I NTER ACTI VE</b>	Specifies whether the application (process) provides a user interface on a desktop that can be used by whoever is logged on when the application is started.
---------------------------------------	--

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

*CommandLine*

String that specifies the command line to execute. The maximum length of

this string is 32K characters.

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. 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:

- ✘ The directory from which the application loaded.
- ✘ The current directory for the parent process.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

Note: If a fully qualified PATH to the program to execute on the remote computer is not given, then you must insure that the PATH variable on the remote computer is setup correctly (to find the program).

Because the "System Environment Variables" will show-through for all users, it is recommended that the "System Environment Variables (PATH)" be configured correctly on the remote computer.

On Windows 2000, you can setup the "System Environment Variables" from the *Control Panel/System/Advanced/Environment Variables/System Variables*.

#### **Returns:**

A positive integer if successful.

A return value of -1 indicates an error (the new process is not started).

If the process is successfully created, the return value is a positive integer that represents the AT Service "Job ID". This Job Id is only valid while the job is running.

#### **Remarks:**

The NetCreateProcess function creates a one-time AT Service job that executes immediately.

The primary benefit of using NetCreateProcess over [NetScheduleJobAdd](#), is that you can use the rich scheduling syntax of TaskGhost to invoke the

call instead of being restricted to the limited AT Service scheduling capabilities.

Things one might consider when deciding whether to use NetCreateProcess or NetScheduleJobAdd might be:

- ✘ Will TaskGhost be running during those times when the job needs to run, and therefore be available to schedule it?
- ✘ Does the schedule require the rich scheduling capabilities of TaskGhost?

If you answered YES to both of the above, then NetCreateProcess is a better choice.

Only members of the Administrators local group can successfully execute the NetCreateProcess function to schedule a job on a remote server.

**This script (NetCreateProcess.vbs) shows how to call NetCreateProcess and how to capture the return code.**

```
Sub main(args)
    Dim server
    Dim commandline
    " set these in your environment, or here in the script
    " If you're running as a service, you will need to set these
    " as SYSTEM variables
    domain = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT,
"SERVER")
    commandline = args
    Dim exit_code
    exit_code = TGCtrl.NetCreateProcess(0,server,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub
```

#### Requirements:

Version 1.0

#### See Also:

[NetCreateProcessAsUser](#), [CreateProcess](#), [NetScheduleJobAdd](#), [NetScheduleJobAddAsUser](#),

NetCreateProcess.vbs in the TaskGhost\Scripts directory.

## NetScheduleJobAddAsUser

### NetScheduleJobAsUser

The NetScheduleJobAsUser function schedules a process to run on a local or remote computer. The NetScheduleJobAsUser function is similar to the [NetScheduleJobAdd](#) function, except that the new process runs in the security context of the user represented by the *user* parameter. By default, the new process is non-interactive, that is, it runs on a desktop that is not visible and cannot receive user input.

The NetScheduleJobAsUser function submits a job to the Windows AT Service to run on the specified computer. This function requires that the Windows schedule service be started at the computer to which the job is submitted.

The NetScheduleJobAsUser function sets the AT Service account name and password before creating the process. The AT Service account name and password are used as the credentials for scheduled jobs created with NetScheduleJobAsUser. NetScheduleJobAsUser impersonates the caller. The caller must be a member of the Administrators group for this function to succeed.

### **Long** NetScheduleJobAsUser (flags, user, server, password, CronSpec, CommandLine)

#### Arguments:

*flags*

The flags parameter modify the default behavior of NetScheduleJobAsUser.

The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the **Or** operator.

<b>NCP_I NTER ACTI VE</b>	Specifies whether the application (process) provides a user interface on a desktop that can be used by whoever is logged on when the application is started.
---------------------------------------	--

*user*

String that specifies the name of the user. This is the name of the user account to log on to.

The user account must have Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

*password*

String that specifies the clear-text password for the user account specified by *user*. For more information about protecting passwords, see [Handling Passwords](#).

*CronSpec*

The Date/Time schedule for the job.

This is a restricted version of TaskGhost's full cron syntax.

Please see the section on scheduling jobs using [cron syntax](#).

*CommandLine*

String that specifies the command line to execute. The maximum length of this string is 32K characters.

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. 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:

- ✘ The directory from which the application loaded.
- ✘ The current directory for the parent process.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

Note: If a fully qualified PATH to the program to execute on the remote computer is not given, then you must insure that the PATH variable on the remote computer is setup correctly (to find the program).

Because the "System Environment Variables" will show-through for all users, it is recommended that the "System Environment Variables

(PATH)" be configured correctly on the remote computer.  
On Windows 2000, you can setup the "System Environment Variables" from the *Control Panel/System/Advanced/Environment Variables/System Variables*.

### Returns:

A positive integer if successful.

A return value of -1 indicates an error (the new process is not started).

If the process is successfully created, the return value is a positive integer that represents the AT Service "Job ID".

This Job ID can be passed directly to [NetScheduleJobDel](#) and [NetScheduleJobGetInfo](#).

### Remarks:

The NetScheduleJobAdd function creates an AT Service job that executes based on the supplied *CronSpec*.

Only members of the Administrators local group can successfully execute the NetScheduleJobAsUser function to schedule a job on a remote server.

**This script (NetScheduleJobAsUser.vbs) shows how to call NetScheduleJobAsUser and how to capture the return code.**

```
Sub main(args)
    Dim user
    Dim server
    Dim password
    Dim commandline
    " set these in your environment, or here in the script
    " If you're running as a service, you will need to set these
    " as SYSTEM variables
    user = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT, "USER")
    server = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT,
"SERVER")
    password = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT Or
TSE_ENV_CRYPT, "PASSWORD")
    commandline = args
    Dim exit_code
    exit_code =
TGCtrl.NetScheduleJobAsUser(0,user,server,password,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub
```

### Requirements:

### Version 1.1

The *user* and *password* parameters are currently ignored and won't be available until Version 1.1.

#### See Also:

[NetScheduleJobAdd](#), [CreateProcess](#), [NetScheduleJobAdd](#), [NetScheduleJobAddAsUser](#),

NetScheduleJobAsUser.vbs in the TaskGhost\Scripts directory.

## NetScheduleJobAdd

### NetScheduleJobAdd

The NetScheduleJobAdd function schedules a process to run on a local or remote computer. The new process then runs the specified executable file. By default, the new process is non-interactive, that is, it runs on a desktop that is not visible and cannot receive user input. Also, by default, the new process inherits the environment of the calling process, rather than the environment associated with the specified user.

The NetScheduleJobAdd function submits a job to the Windows AT Service to run on the specified computer. This function requires that the Windows schedule service be started at the computer to which the job is submitted.

### Long NetScheduleJobAdd (flags, server, CronSpec, CommandLine)

#### Arguments:

*flags*

The flags parameter modify the default behavior of NetScheduleJobAdd. The *flags* parameter can be 0 to assume all defaults, or one or more of the following flags combined with the **Or** operator.

<b>NCP_I NTER ACTI VE</b>	Specifies whether the application (process) provides a user interface on a desktop that can be used by whoever is logged on when the application is started.
---------------------------------------	--

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

*CronSpec*

The Date/Time schedule for the job.

This is a restricted version of TaskGhost's full cron syntax. Please see the section on scheduling jobs using [cron syntax](#).

#### *CommandLine*

String that specifies the command line to execute. The maximum length of this string is 32K characters.

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. 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:

- ✘ The directory from which the application loaded.
- ✘ The current directory for the parent process.
- ✘ The 32-bit Windows system directory.
- ✘ The 16-bit Windows system directory.
- ✘ The Windows directory.
- ✘ The directories that are listed in the PATH environment variable.

Note: If a fully qualified PATH to the program to execute on the remote computer is not given, then you must insure that the PATH variable on the remote computer is setup correctly (to find the program).

Because the "System Environment Variables" will show-through for all users, it is recommended that the "System Environment Variables (PATH)" be configured correctly on the remote computer.

On Windows 2000, you can setup the "System Environment Variables" from the *Control Panel/System/Advanced/Environment Variables/System Variables*.

#### **Returns:**

A positive integer if successful.

A return value of -1 indicates an error (the new process is not started).

If the process is successfully created, the return value is a positive integer that represents the AT Service "Job ID".

#### **Remarks:**

The NetScheduleJobAdd function creates an AT Service job that executes based on the supplied *CronSpec*.

Only members of the Administrators local group can successfully execute the NetScheduleJobAdd function to schedule a job on a remote server.

**This script (NetScheduleJobAdd.vbs) shows how to call NetScheduleJobAdd and how to capture the return code.**

```
Sub main(args)
    Dim server
    Dim commandline
    " set these in your environment, or here in the script
    " If you're running as a service, you will need to set these
    " as SYSTEM variables
    server = TGCtrl.GetEnvironmentVariable (TSE_ENV_PERSISTENT,
"SERVER")
    commandline = args
    Dim exit_code
    exit_code = TGCtrl.NetScheduleJobAdd(0,server,commandline)
    If exit_code = -1 Then
        TGCtrl.Print("Process failed to start!")
    Else
        TGCtrl.Print("Process returned " & exit_code)
    End If
End Sub
```

### Requirements:

Version 1.0

### See Also:

[NetCreateProcessAsUser](#), [CreateProcess](#), [NetScheduleJobAdd](#), [NetScheduleJobAddAsUser](#),  
NetScheduleJobAdd.vbs in the TaskGhost\Scripts directory.

## NetScheduleJobDel

### NetScheduleJobDel

The NetScheduleJobDel function deletes a range of jobs queued to run at a computer. This function requires that the schedule service be started at the computer to which the job deletion request is being sent.

**Bool** NetScheduleJobDel (**server**, **MinJobId**, **MaxJobId**)

### Arguments:

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the

remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

*MinJobId*

Specifies a minimum job identifier. Jobs with a job identifier smaller than *MinJobId* will not be deleted.

*MaxJobId*

Specifies a maximum job identifier. Jobs with a job identifier larger than *MaxJobId* will not be deleted.

**Returns:**

If the function succeeds, the return value is [True](#).

If the function fails, the return value is [False](#).

**Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobDel function on a remote server.

Call the [NetScheduleJobEnum](#) function to retrieve the job identifier for one or more scheduled jobs. You can also use the job identifier returned from [NetScheduleJobAdd](#).

The NetScheduleJobDel function deletes all jobs whose job identifiers are in the range *MinJobId* through *MaxJobId*.

To delete all scheduled jobs at the server, you can call NetScheduleJobDel specifying *MinJobId* equal to 0 and *MaxJobId* equal to - 1. To delete one job, specify the job's identifier for both the *MinJobId* parameter and the *MaxJobId* parameter.

**Requirements:**

Version 1.0

**See Also:**

[NetScheduleJobEnum](#), [NetScheduleJobAdd](#)

**NetScheduleJobGetInfo**

**NetScheduleJobGetInfo**

The NetScheduleJobGetInfo function retrieves information about a particular job queued on a specified computer. This function requires that the schedule service be started.

## **Handle** NetScheduleJobGetInfo (server, JobId)

### **Arguments:**

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

*JobId*

Specifies a value that indicates the identifier of the job for which to retrieve information.

### **Returns:**

If the function succeeds, the return value is a handle to the info object.

If the function fails, the return value is zero.

### **Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobGetInfo function on a remote server.

Once you have a handle to the info object, you can pass it to any of the following functions: [NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoError](#), [NetScheduleJobGetInfoCommand](#), [NetScheduleJobGetInfoJobId](#)

Call [NetScheduleJobEnum](#) to enumerate all jobs and to retrieve their associated info objects.

Use the [FreeHandle](#) to close this handle and free any associated resources.

### **Requirements:**

Version 1.0

### **See Also:**

[NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoError](#), [NetScheduleJobGetInfoCommand](#), [NetScheduleJobGetInfoJobId](#), [NetScheduleJobEnum](#)

NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **NetScheduleJobGetInfoTime**

### **NetScheduleJobGetInfoTime**

The NetScheduleJobGetInfoTime function retrieves information about a particular job queued on a specified computer. This function requires that the schedule service be started.

## **String** NetScheduleJobGetInfoTime (hInfoObject)

### **Arguments:**

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

### **Returns:**

If the function succeeds, the return value is the string representing the Date/Time when this job will next fire.

If the function fails, the return value is [vbNullString](#).

### **Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobGetInfoTime function on a remote server.

Call [NetScheduleJobGetInfo](#) to retrieve an info handle from a job identifier.

Call [NetScheduleJobEnum](#) to enumerate all jobs and to retrieve their associated info objects.

### **Requirements:**

Version 1.0

### **See Also:**

[NetScheduleJobGetInfoError](#), [NetScheduleJobGetInfoCommand](#),

[NetScheduleJobGetInfoJobId](#), [NetScheduleJobEnum](#)

NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **NetScheduleJobGetInfoError**

### **NetScheduleJobGetInfoError**

The NetScheduleJobGetInfoError function retrieves information about a particular job queued on a specified computer. This function requires that the schedule service be started.

## **Bool** NetScheduleJobGetInfoError (hInfoObject)

### **Arguments:**

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

### **Returns:**

[True](#) if the specified job failed the last time it tried to run, [False](#) otherwise.

### **Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobGetInfoError function on a remote server.

Call [NetScheduleJobGetInfo](#) to retrieve an info handle from a job identifier.

Call [NetScheduleJobEnum](#) to enumerate all jobs and to retrieve their associated info objects.

**Requirements:**

Version 1.0

**See Also:**

[NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoCommand](#),  
[NetScheduleJobGetInfoJobId](#), [NetScheduleJobEnum](#)  
NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **NetScheduleJobGetInfoCommand**

### **NetScheduleJobGetInfoCommand**

The NetScheduleJobGetInfoCommand function retrieves information about a particular job queued on a specified computer. This function requires that the schedule service be started.

#### **String NetScheduleJobGetInfoCommand (hInfoObject)**

**Arguments:**

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

**Returns:**

The string representing the command-line to execute or [vbNullString](#) if the command fails.

**Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobGetInfoCommand function on a remote server.

Call [NetScheduleJobGetInfo](#) to retrieve an info handle from a job identifier.

Call [NetScheduleJobEnum](#) to enumerate all jobs and to retrieve their associated info objects.

**Requirements:**

Version 1.0

**See Also:**

[NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoError](#),  
[NetScheduleJobGetInfoJobId](#), [NetScheduleJobEnum](#)  
NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **NetScheduleJobGetInfoJobId**

### **NetScheduleJobGetInfoJobId**

The NetScheduleJobGetInfoJobId function retrieves information about a particular job queued on a specified computer. This function requires that the schedule service be started.

### **Long NetScheduleJobGetInfoJobId (hInfoObject)**

#### **Arguments:**

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

#### **Returns:**

A positive integer representing the job identifier or -1 if the command fails.

#### **Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobGetInfoJobId function on a remote server.

Call [NetScheduleJobGetInfo](#) to retrieve an info handle from a job identifier.

Call [NetScheduleJobEnum](#) to enumerate all jobs and to retrieve their associated info objects.

#### **Requirements:**

Version 1.0

#### **See Also:**

[NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoError](#),  
[NetScheduleJobGetInfoCommand](#), [NetScheduleJobEnum](#)  
NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## **NetScheduleJobEnum**

### **NetScheduleJobEnum**

The NetScheduleJobEnum function lists the jobs queued on a specified computer. This function requires that the schedule service be started.

### **Handle NetScheduleJobEnum (server)**

#### **Arguments:**

*server*

Pointer to a constant string that specifies the DNS or NetBIOS name of the remote server on which the function is to execute. If this parameter is ".", the local computer is used.

This string must begin with \\.

**Returns:**

If the function succeeds, the return value is a handle to the info object.  
If the function fails, the return value is zero.

**Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobEnum function on a remote server.

Pass the handle returned to [NetScheduleJobEnumGet](#) to retrieve info objects for individual jobs.

**Known Problem:** Microsoft has a bug in their implementation of NetScheduleJobEnum.

The problem is that NetScheduleJobEnum will fail to return all jobs if the number of jobs exceeds some threshold (maybe 16 or so.)

Please read the following for [more information](#).

**Requirements:**

Version 1.0

**See Also:**

[NetScheduleJobEnumCount](#), [NetScheduleJobEnumGet](#),  
NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

**NetScheduleJobEnumCount****NetScheduleJobEnumCount**

The NetScheduleJobEnumCount function returns the number of info objects (jobs) returned in an enumeration from [NetScheduleJobEnum](#). This function requires that the schedule service be started.

**Long NetScheduleJobEnumCount (hInfoObject)****Arguments:**

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

**Returns:**

Returns an integer representing the number of jobs in the enumeration or -1 if the command fails.

**Remarks:**

Only members of the Administrators local group can successfully execute the NetScheduleJobEnumCount function on a remote server.

**Requirements:**

Version 1.0

**See Also:**

[NetScheduleJobEnum](#), [NetScheduleJobGetInfoTime](#),  
[NetScheduleJobGetInfoError](#), [NetScheduleJobGetInfoCommand](#),  
[NetScheduleJobGetInfoJobId](#), NetScheduleJobEnum.vbs in the TaskGhost\Scripts  
directory.

## NetScheduleJobEnumGet

### NetScheduleJobEnumGet

The NetScheduleJobEnumGet function returns an info object for an individual job returned in an enumeration from [NetScheduleJobEnum](#). This function requires that the schedule service be started.

#### Handle NetScheduleJobEnumGet (hInfoObject, index)

##### Arguments:

*hInfoObject*

Specifies an info handle from which to retrieve the specified information.

*index*

1 based index of the desired job within the enumeration.

##### Returns:

If the function succeeds, the return value is a handle to the info object.  
If the function fails, the return value is zero.

##### Remarks:

Only members of the Administrators local group can successfully execute the NetScheduleJobEnumGet function on a remote server.

**Note:** The index is not the job identifier.

To get the job identifier, call [NetScheduleJobGetInfoJobId](#).

Handles returned from NetScheduleJobEnumGet can be passed directly to:

- ✘ [NetScheduleJobGetInfoTime](#)
- ✘ [NetScheduleJobGetInfoError](#)
- ✘ [NetScheduleJobGetInfoCommand](#)
- ✘ [NetScheduleJobGetInfoJobId](#)

Use the [FreeHandle](#) to close this handle and free any associated resources.

##### Requirements:

Version 1.0

##### See Also:

[NetScheduleJobEnum](#), [NetScheduleJobEnumCount](#),  
[NetScheduleJobGetInfoTime](#), [NetScheduleJobGetInfoError](#),  
[NetScheduleJobGetInfoCommand](#), [NetScheduleJobGetInfoJobId](#),  
NetScheduleJobEnum.vbs in the TaskGhost\Scripts directory.

## Windows and Messaging

### DispatchMessage

#### DispatchMessage

The DispatchMessage function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the [GetMessage](#) function.

#### Long DispatchMessage (hMsgObject)

##### Arguments:

*hMsgObject*  
Specifies the message object to dispatch.

##### Returns:

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored..

##### Remarks:

Scripts that choose to process messages, must call DispatchMessage for all messages not explicitly handled..

##### Requirements:

Version 1.0

##### See Also:

[GetMessage](#), [GetMessageId](#),  
Schedule.vbs in the TaskGhost\Scripts directory.

### GetMessage

#### GetMessage

Retrieve a message from the thread's (script's) message queue.

#### Handle GetMessage (Reserved1, Reserved2)

##### Arguments:

*Reserved1*  
Must be 0.

*Reserved2*  
Must be 0.

**Returns:**

If the function succeeds, the return value is a handle to the message object.  
If the function fails, the return value is zero.

**Remarks:**

Each script is allocated it's own Win32 thread in which to execute.  
Consequently, the script has it's own Win32 message queue.

In general, scripts will not use this function or any of the other messaging functions because they will be short lived and not require their services. It is typically only scripts that live for an extended amount of time that will make use of the GetMessage and other messaging functions.

The TaskGhost system makes use of GetMessage in it's Schedule.vbs to pump messages to the various message handlers. If you are familiar with lower-level Win32 programming, then you will recognize this model. This so called *Message Pump* in Schedule.vbs is responsible for the processing of messages like TIMER, STOP, SUSPEND, etc.

If you are writing a script which will process (pump) messages, you must process the following messages:

- ✘ WM\_TIMER
- ✘ GM\_SUSPEND
- ✘ GM\_RESUME
- ✘ GM\_TIMECHANGE
- ✘ GM\_STOP
- ✘ GM\_QUIT

Please see Schedule.vbs in the TaskGhost\Scripts directory for a complete example of writing a *Message Pump*.

Use the [FreeHandle](#) to close this handle and free any associated resources.

**Requirements:**

Version 1.0

**See Also:**

[PumpMessage](#), [GetMessageId](#), [GetMessageWParam](#), [GetMessageLParam](#)  
Schedule.vbs in the TaskGhost\Scripts directory.

## **GetMessageId**

### **GetMessageId**

The GetMessageId returns the *message* value from the Win32 MSG structure.

### **Long GetMessageId (hMsgObject)**

#### **Arguments:**

*hMsgObject*

*Specifies an info handle from which to retrieve the specified information.*

Returns:

A positive integer representing the message identifier or -1 if the command fails.

#### **Remarks:**

This function is used to retrieve the 'message' of a Windows message. Developers will normally use this identifier to select an appropriate code branch. For instance, when a timer is started for a script, a developer will use the GetMessageId to extract the message identifier to determine if the new message is a WM\_TIMER message. If so, the WM\_TIMER code branch will be executed.

#### **Requirements:**

Version 1.0

#### **See Also:**

[GetMessage](#), [GetMessageWParam](#), [GetMessageLParam](#),  
Schedule.vbs in the TaskGhost\Scripts directory.

## **GetMessageWParam**

### **GetMessageWParam**

The GetMessageWParam returns the *wParam* value from the Win32 MSG structure.

### **Long GetMessageWParam (hMsgObject)**

#### **Arguments:**

*hMsgObject*

*Specifies an info handle from which to retrieve the specified information.*

Returns:

A positive integer representing the wParam value or -1 if the command fails.

**Remarks:**

This function is used to retrieve the 'wParam' of a Windows message. The meaning of this 'wParam' is message dependant. For instance, the wParam of a WM\_TIMER message is the timer's identifier.

**Requirements:**

Version 1.0

**See Also:**

[GetMessage](#), [GetMessageId](#), [GetMessageLParam](#), [Schedule.vbs](#) in the TaskGhost\Scripts directory.

**GetMessageLParam****GetMessageLParam**

The GetMessageLParam returns the *lParam* value from the Win32 MSG structure.

**Long GetMessageLParam (hMsgObject)****Arguments:**

*hMsgObject*

Specifies an info handle from which to retrieve the specified information.

**Returns:**

A positive integer representing the lParam value or -1 if the command fails.

**Remarks:**

This function is used to retrieve the 'lParam' of a Windows message. The meaning of this 'lParam' is message dependant.

**Requirements:**

Version 1.0

**See Also:**

[GetMessage](#), [GetMessageId](#), [GetMessageWParam](#), [Schedule.vbs](#) in the TaskGhost\Scripts directory.

**KillTimer****KillTimer**

The KillTimer function destroys the specified timer.

## **Bool KillTimer (TimerId)**

### **Arguments:**

*TimerId*

Specifies the timer to be destroyed.

### **Returns:**

If the function succeeds, the return value is **True**.

If the function fails, the return value is **False**.

### **Remarks:**

The KillTimer function does not remove WM\_TIMER messages already posted to the message queue.

### **Requirements:**

Version 1.0

### **See Also:**

[SetTimer](#), [GetMessage](#), [GetMessageId](#), [GetMessageWParam](#),  
Schedule.vbs in the TaskGhost\Scripts directory.

## **PumpMessage**

### **PumpMessage**

Retrieve a message from the thread's (script's) message queue, then dispatches it..

## **Bool PumpMessage (Reserved1, Reserved2)**

### **Arguments:**

*Reserved1*

Must be 0.

*Reserved2*

Must be 0.

### **Returns:**

If the script is clear to continue processing, the return value is **True**.

If the script should stop execution, the return value is **False**.

### **Remarks:**

Each script is allocated it's own Win32 thread in which to execute. Consequently, the script has it's own Win32 message queue.

In general, scripts will not use this function or any of the other messaging functions because they will be short lived and not require their services. It

is typically only scripts that live for an extended amount of time that will make use of the PumpMessage and other messaging functions.

If you are writing a script which will run for an extended period of time, but do not wish to construct a message processing loop, you can use the PumpMessage function to handle message processing for you.

You should be aware that using PumpMessage, you will not be able to process WM\_TIMER, GM\_SUSPEND, and GM\_RESUME messages. To process these messages, you will need to call [GetMessage](#).

Any of the following 3 messages will cause PumpMessage to return **False** thus signaling you to exit your script.

✘ GM\_SUSPEND

✘ GM\_STOP

✘ GM\_QUIT

#### **Requirements:**

Version 1.0

#### **See Also:**

[GetMessage](#), [GetMessageId](#), [GetMessageWParam](#), [GetMessageLParam](#)  
Hello.vbs in the TaskGhost\Scripts directory.

## **SetTimer**

### **SetTimer**

The SetTimer function creates a timer with the specified time-out value.

### **Long SetTimer (Elapse)**

#### **Arguments:**

*Elapse*

Specifies the time-out value, in milliseconds.

#### **Returns:**

If the function succeeds the return value is an integer identifying the new timer. An application can pass this value to the [KillTimer](#) function to destroy the timer.

If the function fails to create a timer, the return value is zero.

#### **Remarks:**

A script can process WM\_TIMER messages by including a WM\_TIMER handler in the script's [Message Pump](#).

The `wParam` parameter of the `WM_TIMER` message contains the timer identifier. This identifier can be used to distinguish one timer from another. Use the [GetMessageWParam](#) to retrieve this identifier.

**Requirements:**

Version 1.0

**See Also:**

[KillTimer](#), [GetMessage](#), [GetMessageId](#), [GetMessageWParam](#),

For an example of a *Message Pump* with `WM_TIMER` processing, please see `Schedule.vbs` in the `TaskGhost\Scripts` directory.

## FindWindow

### FindWindow

The **FindWindow** function retrieves a handle to a window whose class name and window name match the specified strings.

### Handle WindWindow (WindowName, ClassName)

**Arguments:**

*WindowName*

String that specifies the window name (the window's title). If this parameter is [vbNullString](#), all window names match.

This string can be a standard UNIX style regular expression.

*ClassName*

String that specifies the class name created by a previous call to the Win32 API **RegisterClass** or **RegisterClassEx** function. If this parameter is [vbNullString](#), all window class names match.

This string can be a standard UNIX style regular expression.

**Returns:**

If the function succeeds, the return value is a handle to the window that has the specified class and window names.

If the function fails, the return value is zero.

**Remarks:**

During window enumeration, **FindWindow** will call the Win32 API `SendMessageTimeout()` to retrieve the window's caption text or *WindowName*. If the target window is busy or hung, **FindWindow** will only wait 1 second before giving up and continuing on to the next window.

**In the following code snippet we use FindWindow and regular expressions to locate either an unsaved notepad.exe document, or a copy that has already been saved to the file "Hello.txt".**

```
" find the window "Untitled - Notepad" or "Hello.*Notepad"  
Dim hWnd  
hWnd = TGCtrl.FindWindow("Untitled - Notepad|Hello.*Notepad", vbNullString)  
  
If hWnd = 0 Then  
    TGCtrl.Print("Cannot find window!")  
    Exit Sub  
Else  
    TGCtrl.Print("Window found.")  
End If
```

### **Requirements:**

Version 1.1

### **See Also:**

[SetForegroundWindow](#),  
SendKeys.vbs in the TaskGhost\Sample Scripts directory.

## **SetForegroundWindow**

### **SetForegroundWindow**

The **SetForegroundWindow** function puts the thread that created the specified window into the foreground and activates the window. Keyboard input is directed to the window, and various visual cues are changed for the user. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

### **Bool SetForegroundWindow (hWindow)**

#### **Arguments:**

*hWindow*

Handle to the window that should be activated and brought to the foreground.

#### **Returns:**

If the function succeeds, the return value is **True**.

If the function fails, the return value is **False**.

#### **Remarks:**

The *hWindow* parameter is usually the result of a call to the [FindWindow](#) function.

These two functions are generally used in setting up a [SendKeys](#) session.

The following code snippet uses the `SetForegroundWindow` function to set the specified window as the foreground window.

```
" set the foreground window
If TGCtrl.SetForegroundWindow(hWnd) = False Then
    TGCtrl.Print("Cannot SetForegroundWindow!")
    Exit Sub
Else
    TGCtrl.Print("SetForegroundWindow Ok.")
End If
```

#### Requirements:

Version 1.1

#### See Also:

[FindWindow](#),

`SendKeys.vbs` in the `TaskGhost\Sample Scripts` directory.

## Environment Variables and Encryption

### GetEnvironmentVariable

#### GetEnvironmentVariable

The `GetEnvironmentVariable` function retrieves the contents of the specified variable from either transient or persistent local storage.

#### String GetEnvironmentVariable (flags, name)

#### Arguments:

*flags*

The *flags* parameter modify the behavior of `GetEnvironmentVariable`.

The *flags* parameter can be one or more of the following flags combined with the `Or` operator.

<b>ENV_</b> <b>TRA</b> <b>NSIE</b> <b>NT</b>	Only retrieves the contents of the specified variable from the environment block of the calling process.
<b>ENV_</b> <b>PERS</b> <b>ISTE</b> <b>NT</b>	Retrieves the contents of the specified variable from the environment block of the calling process if available, otherwise retrieves the contents of the specified variable from the Windows Registry. May be combined with <b>ENV_FORCE</b> .
<b>ENV_</b> <b>FOR</b> <b>CE</b>	If the desired variables contents are in persistent storage, the <b>ENV_FORCE</b> flag can be combined with <b>ENV_PERSISTENT</b> flag to force the fetch from persistent storage ignoring the transient store.

<b>ENV_CRYPTO</b>	Decrypt the variable's contents.
-------------------	----------------------------------

*name*

Pointer to a string that specifies the name of the environment variable.

**Returns:**

A string representing the variable's value or [vbNullString](#) if the variable does not exist.

**Remarks:**

Transient storage is simply the environment block of the calling process and is equivalent to the normal operation of the Win32 function of the same name.

Persistent storage is accomplished by storing the variable's name and contents in the Windows Registry. This is especially useful for variables such as *user names, passwords, and server names*.

For more information about protecting passwords, see [Handling Passwords](#).

When storing an environment variable with [SetEnvironmentVariable](#) to persistent storage, the transient store is also updated. This is done for speed, because the next fetch from persistent storage will first look in the transient, or cache store.

It is however possible that a value may exist in the persistent store that is preferred over the transient store. This will usually happen when the variable is one of the standard environment variables that are created by the operating system. For example; if you store a copy of the PATH environment variable in the persistent store, then reboot, there will likely be a different copy in the persistent store than in the transient store. In this case you would use the **ENV\_FORCE** flag to force `GetEnvironmentVariable` to read the value from persistent store.

When the **ENV\_FORCE** flag is used in conjunction with `GetEnvironmentVariable`, the transient store is also updated with the data retrieved from the persistent store.

Please see the section on [Handling Passwords](#) for more information on using the **ENV\_CRYPTO** flag.

**Requirements:**

Version 1.0

**See Also:**

## [SetEnvironmentVariable](#),

For more information about protecting passwords, see [Handling Passwords](#).

EnvironmentVariables.vbs in the TaskGhost\Scripts directory.

## **SetEnvironmentVariable**

### **SetEnvironmentVariable**

The SetEnvironmentVariable function sets the contents of the specified variable in either transient or persistent local storage.

### **Bool SetEnvironmentVariable (flags, name, value)**

#### **Arguments:**

*flags*

The flags parameter modify the behavior of SetEnvironmentVariable.

The *flags* parameter can be one or more of the following flags combined with the [Or](#) operator.

<b>ENV_TRANSIENT</b>	Writes the contents of the specified variable to the environment block of the calling process.
<b>ENV_PERSISTENT</b>	Writes the contents of the specified variable to the environment block of the calling process as well as the Windows Registry.
<b>ENV_CRYPT</b>	Encrypt the variable's contents.

*name*

Pointer to a string that specifies the name of the environment variable.

*value*

Pointer to a string that specifies the contents of the environment variable.

An environment variable has a maximum size limit of 32,767 bytes.

If this parameter is [vbNullString](#), the variable is deleted from the appropriate storage.

#### **Returns:**

If the function succeeds, the return value is [True](#).

If the function fails, the return value is [False](#).

#### **Remarks:**

Please see the **remarks** section for [GetEnvironmentVariable](#) for a complete description of the *flags* parameter and use of cryptography in environment variables.

**Requirements:**

Version 1.0

**See Also:**

[GetEnvironmentVariable](#),

For more information about protecting passwords, see [Handling Passwords](#).

EnvironmentVariables.vbs in the TaskGhost\Scripts directory.

## File Transfer Protocol – FTP

### FtpCd

#### FtpCd

Changes to a different working directory on the FTP server.

### **Bool** FtpCd(**hSession**, **RemoteName**)

**Arguments:**

*hSession*

Handle to an FTP session.

*RemoteName*

String that contains the name of the directory to change to on the remote system. This can be either a fully qualified path or a name relative to the current directory.

**Return Value:**

Returns **True** if successful, or **False** otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

**Remarks:**

The *RemoteName* parameter can be either partially or fully qualified file names relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

**Requirements:**

Version 1.1

## See Also:

[FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#),  
"FTP.LASTLINE", "FTP.LASTMSG", "FTP.WINSOCK",  
"FTP.REPLY\_CODE"  
ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpConnect

### FtpConnect

Opens an FTP session with a given site.

### **Handle** FtpConnect(HostName, port, UserId, password)

#### Arguments:

##### *HostName*

String that contains the host name of an Internet server. Alternately, the string can contain the IP number of the site, in ASCII dotted-decimal format (for example, 11.0.1.45).

##### *port*

The TCP/IP port on the server to connect to.

The usual port for FTP operations is port 21.

##### *UserId*

String that contains the name of the user to log on or "anonymous".

##### *password*

String that contains the password to use to log on.

If *UserId* is "anonymous", the password. should be set to the user's e-mail address.

#### Return Value:

Returns a handle to the FTP session if successful, or zero otherwise.

To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

#### Remarks:

After the calling script has finished using the returned handle, it must be closed using the [FreeHandle](#) function.

**The following code demonstrates basic FTP usage by opening a connection to an FTP server and retrieving a remote file and saving it locally.**

```
Sub main(args)

    Dim user          " remote FTP user name
    Dim server        " remote FTP host name
    Dim password      " remote FTP password
    Dim port          " port to connect to (21)
    Dim session       " handle to the FTP session
    Dim binary        " transfer in binary or text?
    Dim LocalFile     " file/directory on local computer
    Dim RemoteFile    " file/directory on remote server

    " Get the user, server, and password from the environment for both
    " security and portability (no hard-coding of this information.)
    " Yes, we hard-coded the file names below, but this *is* a sample!
    user = TGCtrl.GetEnvironmentVariable (ENV_PERSISTENT, "FTP_USER")
    server = TGCtrl.GetEnvironmentVariable (ENV_PERSISTENT, "FTP_SERVER")
    password = TGCtrl.GetEnvironmentVariable (ENV_PERSISTENT Or ENV_CRYPTO,
"FTP_PASSWORD")
    port = 21          " port number. usually 21
    binary = False     " transfer in TEXT mode
    LocalFile = "c:\temp\foo.txt"      " file/directory on local computer
    RemoteFile = "/var/www/html/temp/foo.txt" " file/directory on remote server

    " open your connection to the FTP server
    session = TGCtrl.FtpConnect(server, port, user, password)

    " check for error
    If session = 0 Then
        ShowFtpError(session)
        Exit Sub
    End If

    " DEBUG
    TGCtrl.Print "FTP connect to server " + server + " Ok."
    TGCtrl.Print "lastline: " + TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT,
"FTP.LASTLINE")

    " get a file from your server
    If TGCtrl.FtpGet(session, RemoteFile, LocalFile, binary) = false Then
        ShowFtpError(session)
        Exit Sub
    End If

    " DEBUG
    TGCtrl.Print "FTP get file " + RemoteFile + " Ok."
    TGCtrl.Print "lastline: " + TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT,
"FTP.LASTLINE")

    " close your connection to the FTP server
    If TGCtrl.FreeHandle(session) = false Then
        ShowFtpError(session)
        Exit Sub
    End If
End Sub
```

```

    End If

End Sub

Sub ShowFtpError(session)

    If session = 0 Then
        TGCtrl.Print "FTP open connection failed with error:"
        If len(TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT, "FTP.LASTMSG")) <> 0 Then
            TGCtrl.Print TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT, "FTP.LASTMSG")
        Else
            TGCtrl.Print TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT,
"TG.LAST_ERROR")
        End If
    Else
        TGCtrl.Print "FTP OPERATION failed with error:"
        TGCtrl.Print TGCtrl.GetEnvironmentVariable (ENV_TRANSIENT, "TG.LAST_ERROR")
        TGCtrl.Print "Closing connection."
        TGCtrl.FreeHandle(session)
    End If

End Sub

```

## Requirements:

Version 1.1

## See Also:

[FtpCd](#), [FreeHandle](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#), [FreeHandle](#)  
["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#)  
ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpDel

### FtpDel

Deletes a file stored on the FTP server.

### **Bool** FtpDel(**hSession**, **RemoteName**)

#### Arguments:

*hSession*

Valid FTP handle returned by a previous call to [FtpConnect](#).

*RemoteName*

String that contains the name of the file to delete on the remote system.

#### Return Value:

Returns [True](#) if successful, or [False](#) otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

**Remarks:**

The *RemoteName* parameter can be either a partially or fully qualified file name relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

**Requirements:**

Version 1.1

**See Also:**

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#),  
["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#)  
ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpGet

### FtpGet

Retrieves a file from the FTP server and stores it under the specified file name, creating a new local file in the process.

### **Bool** FtpGet(**hSession**, **RemoteName**, **LocalName**, **bBinary**)

**Arguments:**

*hSession*

Valid handle to an FTP session.

*RemoteName*

String that contains the name of the file to retrieve from the remote system.

*LocalName*

String that contains the name of the file to create on the local system.

*bBinary*

Boolean value specifying if the transfer should be binary ([True](#)), or text ([False](#)).

**Return Value:**

Returns [True](#) if successful, or [False](#) otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

**Remarks:**

If the transfer type is *binary*, the file is downloaded in the same format as it is stored on the server. If the transfer type is *text*, there is an automatic translation of the file data to convert control and formatting characters to local equivalents.

Both *RemoteName* and *LocalName* can be either partially or fully qualified file names relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

**Requirements:**

Version 1.1

**See Also:**

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#),  
["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#)  
ftp.vbs in the TaskGhost\Sample Scripts directory.

**FtpMkDir**

**FtpMkDir**

Creates a new directory on the FTP server.

**Bool FtpMkDir(hSession, RemoteName)**

**Arguments:**

*hSession*

Valid handle returned by a previous call to [FtpConnect](#).

*RemoteName*

String that contains the name of the directory to create on the remote system. This can be either a fully qualified path or a name relative to the current directory.

**Return Value:**

Returns [True](#) if successful, or [False](#) otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

**Remarks:**

The *lpszDirectory* parameter can be either partially or fully qualified file names relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

#### Requirements:

Version 1.1

#### See Also:

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#),  
["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#)  
ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpPut

### FtpPut

Stores a file on the FTP server.

### **Bool** FtpPut(**hSession**, **LocalName**, **RemoteName**, **bBinary**)

#### Arguments:

*hSession*

Valid handle to an FTP session.

*LocalName*

String that contains the name of the file to send from the local system.

*RemoteName*

String that contains the name of the file to create on the remote system.

*bBinary*

Boolean value specifying if the transfer should be binary (**True**), or text (**False**).

#### Return Value:

Returns **True** if successful, or **False** otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

#### Remarks:

If the transfer type is *text*, a translation of the file data converts control and formatting characters to local equivalents.

Both *RemoteName* and *LocalName* can be either partially or fully qualified file names relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

**Requirements:**

Version 1.1

**See Also:**

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#),  
[FtpRmdir](#), [FtpRename](#), [FtpTest](#),  
"[FTP.LASTLINE](#)", "[FTP.LASTMSG](#)", "[FTP.WINSOCK](#)",  
"[FTP.REPLY\\_CODE](#)"  
ftp.vbs in the TaskGhost\Sample Scripts directory.

**FtpRmdir**

**FtpRmdir**

Removes the specified directory on the FTP server.

**Bool FtpRmdir(hSession, RemoteName)**

**Arguments:**

*hSession*

Valid handle to an FTP session.

*RemoteName*

String that contains the name of the directory to remove on the remote system. This can be either a fully qualified path or a name relative to the current directory.

**Return Value:**

Returns **True** if successful, or **False** otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

"[FTP.LASTLINE](#)", "[FTP.LASTMSG](#)", "[FTP.WINSOCK](#)",  
"[FTP.REPLY\\_CODE](#)".

**Remarks:**

The *RemoteName* parameter can be either partially or fully qualified file name relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

**Requirements:**

Version 1.1

## See Also:

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRename](#), [FtpTest](#),  
"FTP.LASTLINE", "FTP.LASTMSG", "FTP.WINSOCK",  
"FTP.REPLY\_CODE"  
ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpRename

### FtpRename

Renames a file stored on the FTP server.

### **Bool** FtpRename(**hSession**, **OldName**, **NewName**)

#### Arguments:

*hSession*

Valid handle to an FTP session.

*OldName*

String that contains the name of the file that will have its name changed on the remote FTP server.

*NewName*

String that contains the new name for the remote file.

#### Return Value:

Returns **True** if successful, or **False** otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

"[FTP.LASTLINE](#)", "[FTP.LASTMSG](#)", "[FTP.WINSOCK](#)",  
"[FTP.REPLY\\_CODE](#)".

#### Remarks:

The *OldName* and *NewName* parameters can be either partially or fully qualified file names relative to the current directory.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

#### Requirements:

Version 1.1

## See Also:

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpTest](#),  
"FTP.LASTLINE", "FTP.LASTMSG", "FTP.WINSOCK",

["FTP.REPLY\\_CODE"](#)

ftp.vbs in the TaskGhost\Sample Scripts directory.

## FtpTest

### FtpTest

Searches the specified directory of the given FTP session for a matching file or directory.

### **Bool** FtpTest(*hSession*, *RemoteName*)

#### Arguments:

*hSession*

Valid handle to an FTP session.

*RemoteName*

String that specifies a valid directory path or file name for the FTP server's file system. The string can contain wildcards, but no blank spaces are allowed. If the value of *RemoteName* is an empty string, it will find the first file in the current directory on the server.

#### Return Value:

Returns **True** if the file or directory exists, or **False** otherwise. To get a specific error message, call [GetEnvironmentVariable\(\)](#) to read one of the FTP related environment variables:

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),  
["FTP.REPLY\\_CODE"](#).

#### Remarks:

Only one **FtpTest** can occur at a time within a given FTP session. The enumerations, therefore, are correlated with the FTP session handle. This is because the FTP protocol allows only a single directory enumeration per session.

See the code snippet for [FtpConnect](#) for an example of an FTP session.

#### Requirements:

Version 1.1

#### See Also:

[FtpCd](#), [FtpClose](#), [FtpConnect](#), [FtpDel](#), [FtpGet](#), [FtpMkDir](#), [FtpPut](#),  
[FtpRmdir](#), [FtpRename](#),

["FTP.LASTLINE"](#), ["FTP.LASTMSG"](#), ["FTP.WINSOCK"](#),

["FTP.REPLY\\_CODE"](#)

ftp.vbs in the TaskGhost\Sample Scripts directory.

