Using the Program

AutoHotkey doesn't do anything on its own; it needs a script to tell it what to do. A script is simply a plain text file with the .ahk filename extension containing instructions for the program, like a configuration file, but much more powerful. A script can do as little as performing a single action and then exiting, but most scripts define a number of hotkeys, with each hotkey followed by one or more actions to take when the hotkey is pressed.

```
#z::Run "https://www.autohotkey.com"  ; Win+Z


^!n::  ; Ctrl+Alt+N
{
   if WinExist("Untitled - Notepad")
      WinActivate
   else
      Run "Notepad"
}
```

Tip: If your browser supports it, you can download any code block (such as the one above) as a script file by clicking the button which appears in the top-right of the code block when you hover your mouse over it.

Table of Contents

Portability of AutoHotkey.exe

Launcher

Dash

New Script

Installation

Run with UI Access

Create a Script

There are a few common ways to create a script file:

In Notepad (or a text editor of your choice), save a file with the .ahk filename extension. On some systems you may need to enclose the name in quotes to ensure the editor does not add another extension (such as .txt).

Be sure to save the file as UTF-8 with BOM if it will contain non-ASCII characters. For details, see the FAQ.

In Explorer, right-click in empty space in the folder where you want to save the script, then select New and AutoHotkey Script. You can then type a name for the script (taking care not to erase the .ahk extension if it is visible).

In the Dash, select New script, type a name for the script (excluding the .ahk extension) and click Create or Edit. The template used to create the script and the location it will be saved can be configured within this window, and set as default if desired.

See Scripting Language for details about how to write a script.

Edit a Script

To open a script for editing, right-click on the script file and select Edit Script. If the script is already running, you can use the Edit function or right-click the script's tray icon and select Edit Script. If you haven't selected a default editor yet, you should be prompted to select one. Otherwise, you can change your default editor via Editor settings in the Dash. Of course, you can always open a text editor first and then open the script as you would any other text file.

After editing a script, you must run or reload the script for the changes to take effect. A running script can usually be reloaded via its tray menu.

Run a Script

With AutoHotkey installed, there are several ways to run a script:

Double-click a script file (or shortcut to a script file) in Explorer.

Call AutoHotkey.exe on the command line and pass the script's filename as a command-line parameter.

After creating the default script, launch AutoHotkey via the shortcut in the Start menu to run it.

If AutoHotkey is pinned to the taskbar or Start menu on Windows 7 or later, recent or pinned scripts can be launched via the program's Jump List.

Most scripts have an effect only while they are running. Use the tray menu or the ExitApp function to exit a script. Scripts are also forced to exit when Windows shuts down. To configure a script to start automatically after the user logs in, the easiest way is to place a shortcut to the script file in the Startup folder.

Scripts can also be compiled; that is, combined together with an AutoHotkey binary file to form a self-contained executable (.exe) file.

Tray Icon

By default, each script adds its own icon to the taskbar notification area (commonly known as the tray).

The tray icon usually looks like this (but the color or letter changes when the script is paused or suspended): H

Right-click the tray icon to show the tray menu, which has the following options by default:

Open - Open the script's main window.

Help - Open the AutoHotkey offline help file.

Window Spy - Displays various information about a window.

Reload Script - See Reload.

Edit Script - See Edit.

Suspend Hotkeys - Suspend or unsuspend hotkeys.

Pause Script - Pause or unpause the script.

Exit - Exit the script.

By default, double-clicking the tray icon shows the script's main window.

The behavior and appearance of the tray icon and menu can be customized:

A_TrayMenu returns a Menu object which can be used to customise the tray menu.

A_IconHidden or the #NoTrayIcon directive can be used to hide (or show) the tray icon.

A_IconTip can be assigned new tooltip text for the tray icon.

TraySetIcon can be used to change the icon.

Main Window

The script's main window is usually hidden, but can be shown via the tray icon or one of the functions listed below to gain access to information useful for debugging the script. Items under the View menu control what the main window displays:

Lines most recently executed - See ListLines.

Variables and their contents - See ListVars.

Hotkeys and their methods - See ListHotkeys.

Key history and script info - See KeyHistory.

Known issue: Keyboard shortcuts for menu items do not work while the script is displaying a message box or other dialog.

The built-in variable A_ScriptHwnd contains the unique ID (HWND) of the script's main window.

Closing this window with WinClose (even from another script) causes the script to exit, but most other methods just hide the window and leave the script running.

Minimizing the main window causes it to automatically be hidden. This is done to prevent any owned windows (such as GUI windows or certain dialog windows) from automatically being minimized, but also has the effect of hiding the main window's taskbar button. To instead allow the main window to be minimized normally, override the default handling with OnMessage. For example:

```
; This prevents the main window from hiding on minimize:
OnMessage 0x0112, PreventAutoMinimize ; WM_SYSCOMMAND = 0x0112
OnMessage 0x0005, PreventAutoMinimize ; WM_SIZE = 0x0005
; This prevents owned GUI windows (but not dialogs) from automatically minimizing:
OnMessage 0x0018, PreventAutoMinimize
Persistent

PreventAutoMinimize(wParam, lParam, uMsg, hwnd) {
   if (uMsg = 0x0112 && wParam = 0xF020 && hwnd = A_ScriptHwnd) { ; SC_MINIMIZE = 0xF020
      WinMinimize
      return 0 ; Prevent main window from hiding.
   }
   if (uMsg = 0x0005 && wParam = 1 && hwnd = A_ScriptHwnd) ; SIZE_MINIMIZED = 1
      return 0 ; Prevent main window from hiding.
   if (uMsg = 0x0018 && lParam = 1) ; SW_PARENTCLOSING = 1
      return 0 ; Prevent owned window from minimizing.
}
```

Main Window Title

The title of the script's main window is used by the #SingleInstance and Reload mechanisms to identify other instances of the same script. Changing the title prevents the script from being identified as such. The default title depends on how the script was loaded:

Loaded From	Title Expression	Example
.ahk file	A_ScriptFullPath " - AutoHotkey v" A_AhkVersion	E:\My Script.ahk - AutoHotkey v1.1.33.09
Main resource (compiled script)	A_ScriptFullPath	E:\My Script.exe
Any other resource	A_ScriptFullPath " - " A_LineFile	E:\My AutoHotkey.exe - *BUILTIN-TOOL.AHK

The following code illustrates how the default title could be determined by the script itself (but the actual title can be retrieved with WinGetTitle):

```
title := A_ScriptFullPath
if !A_IsCompiled
    title .= " - AutoHotkey v" A_AhkVersion
; For the correct result, this must be evaluated by the resource being executed,
; not an #include (unless the #include was merged into the script by Ahk2Exe):
else if SubStr(A_LineFile, 1, 1) = "*" && A_LineFile != "*#1"
    title .= " - " A_LineFile
```

Embedded Scripts

Scripts may be embedded into a standard AutoHotkey .exe file by adding them as Win32 (RCDATA) resources using the Ahk2Exe compiler. To add additional scripts, see the AddResource compiler directive.

An embedded script can be specified on the command line or with #Include by writing an asterisk (*) followed by the resource name. For an integer ID, the resource name must be a hash sign (#) followed by a decimal number.

The program may automatically load script code from the following resources, if present in the file:

ID	Spec	Usage
1	*#1	This is the means by which a compiled script is created from an .exe file. This script is executed automatically and most command line switches are passed to the script instead of being interpreted by the program. External scripts and alternative embedded scripts can be executed by using the /script switch.
2	*#2	If present, this script is automatically "included" before any script that the program loads, and before any file specified with /include.

When the source of the main script is an embedded resource, the program acts in "compiled script" mode, with the exception that A_AhkPath always contains the path of the current executable file (the same as A_ScriptFullPath). For resources other than *#1, the resource specifier is included in the main window's title to support #SingleInstance and Reload.

When referenced from code that came from an embedded resource, A_LineFile contains an asterisk (*) followed by the resource name.

Command Line Usage

See Passing Command Line Parameters to a Script for command line usage, including a list of command line switches which affect the program's behavior.

Portability of AutoHotkey.exe

The file AutoHotkey.exe is all that is needed to launch any .ahk script.

Renaming AutoHotkey.exe also changes which script it runs by default, which can be an alternative to compiling a script for use on a computer without AutoHotkey installed. For instance, MyScript.exe automatically runs MyScript.ahk if a filename is not supplied, but is also capable of running other scripts.

Launcher

The launcher enables the use of v1 and v2 scripts on one system, with a single filename extension, without necessarily giving preference to one version or requiring different methods of launching scripts. It does this by checking the script for clues about which version it requires, and then locating an appropriate exe to run the script.

If the script contains the #Requires directive, the launcher looks for an exe that satisfies the requirement. Otherwise, the launcher optionally checks syntax. That is, it checks for patterns that are valid only in one of the two major versions. Some of the common patterns it may find include:

v1: MsgBox, with comma, MsgBox % "no end percent" and Legacy = assignment.

v1: Multi-line hotkeys without braces or a function definition.

Common directives such as #NoEnv and #If (v1) or #HotIf (v2).

v2: Unambiguous use of continuation by enclosure or end-of-line continuation operators.

v2: Unambiguous use of 'single quotes' or fat arrow => in an expression.

Detection is conservative; if a case is ambiguous, it should generally be ignored.

In any case where detection fails, by default a menu is shown for the user to select a version. This default can be changed to instead launch either v1 or v2.

Known limitations:

Only the main file is checked.

Since it is legal to include a line like /****/ in v1, but */ at line-end only closes comments in v2, the presence of such a line may cause a large portion of the script to be ignored (by both the launcher and the v1 interpreter).

Only syntax is checked, not semantics. For instance, xyz, is invalid in v2, so is assumed to be a valid v1 command. xyz 1 could be a function statement in v2, but is assumed to also be a valid v1 command, and is therefore ignored.

Since the patterns being detected are effectively syntax errors in one version, a script with actual syntax errors or incorrectly mixed syntax might be misidentified.

Note: Declaring the required version with #Requires at the top of the main file eliminates any ambiguity.

Launch Settings

The launcher can be enabled, disabled or configured via the Launch Settings GUI, which can be accessed via the dash.

Run all scripts with a specific interpreter disables the launcher and allows the user to select which exe to use to run all scripts, the traditional way. Be aware that selecting a v1 exe will make it difficult to run any of the support scripts, except via the "AutoHotkey" shortcut in the Start menu.

Auto-detect version when launching script enables the launcher. Additional settings control how the launcher selects which interpreter to use.

Criteria

When multiple interpreters with the same version number are found, the launcher can rank them according to a predetermined or user-defined set of criteria. The criteria can be expressed as a comma-delimited list of substrings, where each substring may be prefixed with "!" to negate a match. A score is calculated based on which substrings matched, with the left-most substring having highest priority.

Substrings are matched in the file's description, with the exception of "UIA", which matches if the filename contains "_UIA".

For example, _H, 64, !ANSI would prefer AutoHotkey_H if available, 64-bit if compatible with the system, and finally Unicode over ANSI.

Although the Launcher Settings GUI presents drop-down lists with options such as "Unicode 32-bit", a list of substrings can be manually entered.

Additional (higher-priority) criteria can be specified on the command line with the /RunWith launcher switch.

Criteria can be specified within the script by using the #Requires directive, either as a requirement (if supported by the target AutoHotkey version), or appended to the directive as a comment beginning with "prefer" and ending with a full stop or line ending. For example:

#Requires AutoHotkey v1.1.35 ; prefer 64-bit, Unicode. More comments.

Run *Launch

The installer registers a hidden shell verb named "launch", which executes the launcher with the /Launch switch. It can be utilized by following this example:

pid := RunWait('*Launch "' PathOfScript '"')

By contrast with the default action for .ahk files:

/Launch causes the process ID (PID) of the newly launched script to be returned as the launcher's exit code, whereas it would normally return the launched script's exit code. Run's OutputVarPID parameter returns the PID of the launcher.

/Launch causes the launcher to exit immediately after launching the script. If /Launch is not used, the launcher generally has to assume that its parent process might be doing something like RunWait(PathOfScript), which wouldn't work as expected if the launcher exited before the launched script.

Command-line Usage

The launcher can be explicitly executed at the command line for cases where .ahk files are not set to use the launcher by default, or for finer control over its behaviour. If the launcher is compiled, its usage is essentially the same as AutoHotkey.exe except for the additional launcher switches. Otherwise, the format for command line use is as follows:

AutoHotkeyUX.exe launcher.ahk [Switches] [Script Filename] [Script Parameters]

Typically full paths and quote marks would be used for the path to AutoHotkeyUX.exe and launcher.ahk, which can be found in the UX subdirectory of the AutoHotkey installation. An appropriate version of AutoHotkey32.exe or AutoHotkey64.exe can be used instead of AutoHotkeyUX.exe (which is just a copy).

Switches can be a mixture of any of the standard switches and the following launcher-only switches:

Switch Meaning

/Launch Causes the launcher to exit immediately after launching the script, instead of waiting in the background for it to terminate. The launcher's exit code is the process ID (PID) of the new script process.

/RunWith criteria Specifies additional criteria for determining which executable to use to launch the script. For example, /RunWith UIA.

/Which

Causes the launcher to identify which interpreter it would use and return it instead of running the script.

The launcher's exit code is the major version number (1 or 2) if identified by #Requires or syntax (if syntax detection is enabled), otherwise 0.

Stdout receives the following UTF-8 strings, each terminated with `n:

The version number. If #Requires was detected, this is whatever number it specified, excluding "v". Otherwise, it is an integer the same as the exit code, unless the version wasn't detected, in which case this is 0 to indicate that the user would have been prompted, or 1 or 2 to indicate the user's preferred version as configured in the Launch Settings.

The path of the interpreter EXE that would be used, if one was found. This is blank if the user would have been prompted or no compatible interpreters were found.

Any additional command-line switches that the launcher would insert, such as /CP65001.

Additional lines may be returned in future.

Dash

The dash provides access to support scripts and documentation. It can be opened via the "AutoHotkey" shortcut in the Start menu after installation, or by directly running UX\ui-dash.ahk from the installation directory. Currently it is little more than a menu containing the following items, but it might be expanded to provide controls for active scripts, or other convenient functions.

New script: Create a new script from a template.

Compile: Opens Ahk2Exe, or offers to automatically download and install it.

Help files (F1): Shows a menu containing help files and online documentation for v1 and v2, and any other CHM files found in the installation directory.

Window spy

Launch settings: Configure the launcher.

Editor settings: Set the default editor for .ahk files.

Note that although the Start menu shortcut launches the dash, if it is pinned to the taskbar (or to the Start menu in Windows 7 or 10), the jump list will include any recent scripts launched with the open, runas or UIAccess shell verbs (which are normally accessed via the Explorer context menu or by double-clicking a file). Scripts can be pinned for easy access.

New Script

The New Script GUI can be accessed via the dash or by right-clicking within a folder in Explorer and selecting New → AutoHotkey Script. It can be used to create a new script file from a preinstalled or user-defined template, and optionally open it for editing.

Right-clicking on a template in the list gives the following options:

Edit template: Open the template in the default editor. If it is a preinstalled template, an editable copy is created instead of opening the original.

Hide template: Adds the template name to a list of templates that will not be shown in the GUI. To unhide a template, delete the corresponding registry value from HKCU\Software\AutoHotkey\New\HideTemplate.

Set as default: Sets the template to be selected by default.

By default, the GUI closes after creating a file unless the Ctrl key is held down.

Additional settings can be accessed via the settings button at the bottom-left of the GUI:

Default to Create: Pressing Enter will activate the Create button, which creates the script and selects it in Explorer.

Default to Edit: Pressing Enter will activate the Edit button, which creates the script and opens it in the default script editor.

Stay open: If enabled, the window will not close automatically after creating a script.

Set folder as default: Sets the current folder as the default location to save scripts. The default location is used if the New Script window is opened directly or via the Dash; it is not used when New Script is invoked via the Explorer context menu.

Open templates folder: Opens the folder where user-defined templates can be stored.

Templates

Template files are drawn from UX\Templates (preinstalled)
and %A_MyDocuments%\AutoHotkey\Templates (user), with a user-defined template overriding any preinstalled template which has the same name. If a file exists at %A_WinDir%\ShellNew\Template.ahk, it is shown as "Legacy" and can be overridden by a user-defined template of that name.

Each template may contain an INI section as follows:

```
/*

[NewScriptTemplate]

Description = Descriptive text

Execute = true|false|1|0

*/
```

If the INI section starts with /* and ends with */ as shown above, it is not included in the created file.

Description is optional. It is shown in the GUI, in addition to the file's name.

Execute is optional. If set to true, the template script is executed with A_Args[1] containing the path of the file to be created and A_Args[2] containing either "Create" or "Edit", depending on which button the

user clicked. The template script is expected to create the file and open it for editing if applicable. If the template script needs to #include other files, they can be placed in a subdirectory to avoid having them shown in the template list.

Installation

This installer and related scripts are designed to permit multiple versions of AutoHotkey to coexist. The installer provides very few options, as most things can be configured after installation. Only the following choices must be made during installation:

Where to install.

Whether to install for all users or the current user.

By default the installer will install to "%A_ProgramFiles%\AutoHotkey" for all users. This is recommended, as the UI Access option requires the program to be installed under Program Files. If the installer is not already running as admin, it will attempt to elevate when the Install button is clicked, as indicated by the shield icon on the button.

Current user installation does not require admin rights, as long as the user has write access to the chosen directory. The default directory for a current user installation is "%LocalAppData%\Programs\AutoHotkey".

Installing with v1

There are two methods of installing v1 and v2 together:

Install v1 first, and then v2. In that case, the v1 files are left in the root of the installation directory, to avoid breaking any external tools or shortcuts that rely on their current path.

Install v1 as an additional version. Running a v1.1.34.03 or later installer gives this option. Alternatively, use the /install switch described below. Each version installs into its own subdirectory.

Running a v1.1.34.02 or older installer (or a custom install with v1.1.34.03 or newer) will overwrite some of the values set in the registry by the v2 installer, such as the version number, uninstaller entry and parts of the file type registration. It will also register the v1 uninstaller, which is not capable of correctly

uninstalling both versions. To re-register v2, re-run any v2 installer or run UX\install.ahk using AutoHotkey32.exe or AutoHotkey64.exe.

Default Version

Unlike a v1 installation, a default version is not selected during installation. Defaults are instead handled more dynamically by the launcher, and can be configured per-user.

Command Line Usage

To directly install to the DESTINATION directory, use /installto or /to (the two switches are interchangeable) as shown below, from within the source directory. Use either a downloaded setup.exe or files extracted from a downloaded zip or other source.

AutoHotkey_setup.exe /installto "%DESTINATION%"

AutoHotkey32.exe UX\install.ahk /to "%DESTINATION%"

To install an additional version from SOURCE (which should be a directory containing AutoHotkey*.exe files), execute the following from within the current installation directory (adjusting the path of AutoHotkey32.exe as needed):

AutoHotkey32.exe UX\install.ahk /install "%SOURCE%"

The full command string for the above is registered as InstallCommand under HKLM\Software\AutoHotkey or HKCU\Software\AutoHotkey, with %1 as the substitute for the source directory. Using this registry value may be more future-proof.

To re-register the current installation:

AutoHotkey32.exe UX\install.ahk

To uninstall:

AutoHotkey32.exe UX\install.ahk /uninstall

Alternatively, read the QuietUninstallString value from one of the following registry keys, and execute it:

HKLM\Microsoft\Windows\CurrentVersion\Uninstall\AutoHotkey

HKCU\Microsoft\Windows\CurrentVersion\Uninstall\AutoHotkey

Use the /silent switch to suppress warning or confirmation dialogs and prevent the Dash from being shown when installation is complete. The following actions may be taken automatically, without warning:

Terminate scripts to allow AutoHotkey*.exe to be overwritten.

Overwrite files that were not previously registered by the installer, or that were modified since registration.

Taskbar Buttons

The v2 installer does not provide an option to separate taskbar buttons. This was previously achieved by registering each AutoHotkey executable as a host app (IsHostApp), but this approach has limitations, and becomes less manageable when multiple versions can be installed. Instead, each script should set the AppUserModelID of its process or windows to control grouping.

Run with UI Access

When installing under Program Files, the installer creates an additional set of AutoHotkey exe files that can be used to work around some common UAC-related issues. These files are given the "_UIA.exe" suffix. When one of these UIA.exe files is used by an administrator to run a script, the script is able to interact with windows of programs that run as admin, without the script itself running as admin.

The installer does the following:

Copies each AutoHotkey*.exe to AutoHotkey*_UIA.exe.

Sets the uiAccess attribute in each UIA.exe file's embedded manifest.

Creates a self-signed digital certificate named "AutoHotkey" and signs each UIA.exe file.

Registers the UIAccess shell verb, which appears in Explorer's context menu as "Run with UI access". By default this executes the launcher, which tries to select an appropriate UIA.exe file to run the script.

The launcher can also be configured to run v1 scripts, v2 scripts or both with UI Access by default, but this option has no effect if a UIA.exe file does not exist for the selected version and build.

Scripts which need to run other scripts with UI access can simply Run the appropriate UIA.exe file with the normal command line parameters. Alternatively, if the UIAccess shell verb is registered, it can be used via Run. For example: Run '*UIAccess "Script.ahk"'

Known limitations:

UIA is only effective if the file is in a trusted location; i.e. a Program Files sub-directory.

UIA.exe files created on one computer cannot run on other computers without first installing the digital certificate which was used to sign them.

UIA.exe files cannot be started via CreateProcess due to security restrictions. ShellExecute can be used instead. Run tries both.

UIA.exe files cannot be modified, as it would invalidate the file's digital signature.

Because UIA programs run at a different "integrity level" than other programs, they can only access objects registered by other UIA programs. For example, ComObjActive("Word.Application") will fail because Word is not marked for UI Access.

The script's own windows can't be automated by non-UIA programs/scripts for security reasons.

Running a non-UIA script which uses a mouse hook (even as simple as InstallMouseHook) may prevent all mouse hotkeys from working when the mouse is pointing at a window owned by a UIA script, even hotkeys implemented by the UIA script itself. A workaround is to ensure UIA scripts are loaded last.

UIA prevents the Gui +Parent option from working on an existing window if the new parent is always-on-top and the child window is not.

For more details, see Enable interaction with administrative programs on the archive forum.

Concepts and Conventions

This document covers some general concepts and conventions used by AutoHotkey, with focus on explanation rather than code. The reader is not assumed to have any prior knowledge of scripting or programming, but should be prepared to learn new terminology.

For more specific details about syntax, see Scripting Language.

Table of Contents

String Encoding

Pure Numbers

Names

References to Objects

Values

A value is simply a piece of information within a program. For example, the name of a key to send or a program to run, the number of times a hotkey has been pressed, the title of a window to activate, or whatever else has some meaning within the program or script.

AutoHotkey supports these types of values:

Strings (text)

Numbers (integers and floating-point numbers)

Objects

The Type function can be used to determine the type of a value.

Some other related concepts:

Boolean

Nothing

Strings

A string is simply text. Each string is actually a sequence or string of characters, but can be treated as a single entity. The length of a string is the number of characters in the sequence, while the position of a character in the string is merely that character's sequential number. By convention in AutoHotkey, the first character is at position 1.

Numeric strings: A string of digits (or any other supported number format) is automatically interpreted as a number when a math operation or comparison requires it.

How literal text should be written within the script depends on the context. For instance, in an expression, strings must be enclosed in quotation marks. In directives (excluding #HotIf) and auto-replace hotstrings, quotation marks are not needed.

For a more detailed explanation of how strings work, see String Encoding.

Numbers

AutoHotkey supports these number formats:

Decimal integers, such as 123, 00123 or -1.

Hexadecimal integers, such as 0x7B, 0x007B or -0x1.

Decimal floating-point numbers, such as 3.14159.

Hexadecimal numbers must use the 0x or 0X prefix, except where noted in the documentation. This prefix must be written after the + or - sign, if present, and before any leading zeroes. For example, 0x001 is valid, but 000x1 is not.

Numbers written with a decimal point are always considered to be floating-point, even if the fractional part is zero. For example, 42 and 42.0 are usually interchangeable, but not always. Scientific notation is also recognized (e.g. 1.0e4 and -2.1E-4), but always produces a floating-point number even if no decimal point is present.

The decimal separator is always a dot, even if the user's regional settings specify a comma.

When a number is converted to a string, it is formatted as decimal. Floating-point numbers are formatted with full precision (but discarding redundant trailing zeroes), which may in some cases reveal their inaccuracy. Use the Format function to produce a numeric string in a different format. Floating-point numbers can also be formatted by using the Round function.

For details about the range and accuracy of numeric values, see Pure Numbers.

Boolean

A boolean value can be either true or false. Boolean values are used to represent anything that has exactly two possible states, such as the truth of an expression. For example, the expression (x <= y) is true when x has lesser or equal value to y. A boolean value could also represent yes or no, on or off, down or up (such as for GetKeyState) and so on.

AutoHotkey does not have a specific boolean type, so it uses the integer value 0 to represent false and 1 to represent true. When a value is required to be either true or false, a blank or zero value is considered false and all other values are considered true. (Objects are always considered true.)

The words true and false are built-in variables containing 1 and 0. They can be used to make a script more readable.

Nothing

AutoHotkey does not have a value which uniquely represents nothing, null, nil or undefined, as seen in other languages.

Instead of producing a "null" or "undefined" value, any attempt to read a variable, property, array element or map item which has no value causes an UnsetError to be thrown. This allows errors to be identified more easily than if a null value was implicitly allowed to propagate through the code. See also: Uninitialized Variables.

A function's optional parameters can be omitted when the function is called, in which case the function may change its behaviour or use a default value. Parameters are usually omitted by literally omitting them from the code, but can also be omitted explicitly or conditionally by using the unset keyword. This special signal can only be propagated explicitly, with the maybe operator (var?). An unset parameter automatically receives its default value (if any) before the function executes.

Mainly for historical reasons, an empty string is sometimes used wherever a null or undefined value would be used in other languages, such as for functions which have no explicit return value.

If a variable or parameter is said to be "empty" or "blank", that usually means an empty string (a string of zero length). This is not the same as omitting a parameter, although it may have the same effect in some cases.

Objects

The object is AutoHotkey's composite or abstract data type. An object can be composed of any number of properties (which can be retrieved or set) and methods (which can be called). The name and effect of each property or method depends on the specific object or type of object.

Objects have the following attributes:

Objects are not contained; they are referenced. For example, alpha := [] creates a new Array and stores a reference in alpha. bravo := alpha copies the reference (not the object) to bravo, so both refer to the same object. When an array or variable is said to contain an object, what it actually contains is a reference to the object.

Two object references compare equal only if they refer to the same object.

Objects are always considered true when a boolean value is required, such as in if obj, !obj or obj ? x : y.

Each object has a unique address (location in memory), which can be retrieved by the ObjPtr function, but is typically not used directly. This address uniquely identifies the object, but only until the object is freed.

In some cases when an object is used in a context where one was not expected, it might be treated as an empty string. For example, MsgBox(myObject) shows an empty MsgBox. In other cases, a TypeError may be thrown (and this should become the norm in future).

Note: All objects which derive from Object have additional shared behaviour, properties and methods.

Some ways that objects are used include:

To contain a collection of items or elements. For example, an Array contains a sequence of items, while a Map associates keys with values. Objects allow a group of values to be treated as one value, to be assigned to a single variable, passed to or returned from a function, and so on.

To represent something real or conceptual. For example: a position on the screen, with X and Y properties; a contact in an address book, with Name, PhoneNumber, EmailAddress and so on. Objects can be used to represent more complex sets of information by combining them with other objects.

To encapsulate a service or set of services, allowing other parts of the script to focus on a task rather than how that task is carried out. For example, a File object provides methods to read data from a file or write data to a file. If a script function which writes information to a file accepts a File object as a parameter, it needs not know how the file was opened. The same function could be reused to write information to some other target, such as a TCP/IP socket or WebSocket (via user-defined objects).

A combination of the above. For example, a Gui represents a GUI window; it provides a script with the means to create and display a graphical user interface; it contains a collection of controls, and provides information about the window via properties such as Title and FocusedCtrl.

The proper use of objects (and in particular, classes) can result in code which is modular and reusable. Modular code is usually easier to test, understand and maintain. For instance, one can improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections. Reusable code saves time, by avoiding the need to write and test code for the same or similar tasks over and over.

Object Protocol

This section builds on these concepts which are covered in later sections: variables, functions

Objects work through the principle of message passing. You don't know where an object's code or variables actually reside, so you must pass a message to the object, like "give me foo" or "go do bar", and rely on the object to respond to the message. Objects in AutoHotkey support the following basic messages:

Get a property.

Set a property, denoted by :=.

Call a method, denoted by ().

A property is simply some aspect of the object that can be set and/or retrieved. For example, Array has a Length property which corresponds to the number of elements in the array. If you define a property, it

can have whatever meaning you want. Generally a property acts like a variable, but its value might be calculated on demand and not actually stored anywhere.

Each message contains the following, usually written where the property or method is called:

The name of the property or method.

Zero or more parameters which may affect what action is carried out, how a value is stored, or which value is returned. For example, a property might take an array index or key.

For example:

myObj.methodName(arg1)

value := myObj.propertyName[arg1]

An object may also have a default property, which is invoked when square brackets are used without a property name. For example:

value := myObj[arg1]

Generally, Set has the same meaning as an assignment, so it uses the same operator:

myObj.name := value

myObj.name[arg1, arg2, …, argN] := value

myObj[arg1, arg2, …, argN] := value

Variables

A variable allows you to use a name as a placeholder for a value. Which value that is could change repeatedly during the time your script is running. For example, a hotkey could use a variable press_count to count the number of times it is pressed, and send a different key whenever press_count is a multiple of 3 (every third press). Even a variable which is only assigned a value once can be useful. For example, a WebBrowserTitle variable could be used to make your code easier to update when and if you were to change your preferred web browser, or if the title or window class changes due to a software update.

In AutoHotkey, variables are created simply by using them. Each variable is not permanently restricted to a single data type, but can instead hold a value of any type: string, number or object. Attempting to read a variable which has not been assigned a value is considered an error, so it is important to initialize variables.

A variable has three main aspects:

The variable's name.

The variable itself.

The variable's value.

Certain restrictions apply to variable names - see Names for details. In short, it is safest to stick to names consisting of ASCII letters (which are case insensitive), digits and underscore, and to avoid using names that start with a digit.

A variable name has scope, which defines where in the code that name can be used to refer to that particular variable; in other words, where the variable is visible. If a variable is not visible within a given scope, the same name can refer to a different variable. Both variables might exist at the same time, but only one is visible to each part of the script. Global variables are visible in the "global scope" (that is, outside of functions), and can be read by functions by default, but must be declared if they are to be assigned a value inside a function. Local variables are visible only inside the function which created them.

A variable can be thought of as a container or storage location for a value, so you'll often find the documentation refers to a variable's value as the contents of the variable. For a variable x := 42, we can also say that the variable x has the number 42 as its value, or that the value of x is 42.

It is important to note that a variable and its value are not the same thing. For instance, we might say "myArray is an array", but what we really mean is that myArray is a variable containing a reference to an array. We're taking a shortcut by using the name of the variable to refer to its value, but "myArray" is really just the name of the variable; the array object doesn't know that it has a name, and could be referred to by many different variables (and therefore many names).

Uninitialized Variables

To initialize a variable is to assign it a starting value. A variable which has not yet been assigned a value is uninitialized (or unset for short). Attempting to read an uninitialized variable is considered an error. This helps to detect errors such as mispelled names and forgotten assignments.

IsSet can be used to determine whether a variable has been initialized, such as to initialize a global or static variable on first use.

A variable can be un-set by combining a direct assignment (:=) with the unset keyword or the maybe (var?) operator. For example: Var := unset, Var1 := (Var2?).

The or-maybe operator (??) can be used to provide a default value when a variable lacks a value. For example, MyVar ?? "Default" is equivalent to IsSet(MyVar) ? MyVar : "Default".

Built-in Variables

A number of useful variables are built into the program and can be referenced by any script. Except where noted, these variables are read-only; that is, their contents cannot be directly altered by the script. By convention, most of these variables start with the prefix A_, so it is best to avoid using this prefix for your own variables.

Some variables such as A_KeyDelay and A_TitleMatchMode represent settings that control the script's behavior, and retain separate values for each thread. This allows subroutines launched by new threads (such as for hotkeys, menus, timers and such) to change settings without affecting other threads.

Some special variables are not updated periodically, but rather their value is retrieved or calculated whenever the script references the variable. For example, A_Clipboard retrieves the current contents of the clipboard as text, and A_TimeSinceThisHotkey calculates the number of milliseconds that have elapsed since the hotkey was pressed.

Related: list of built-in variables.

Environment Variables

Environment variables are maintained by the operating system. You can see a list of them at the command prompt by typing SET then pressing Enter.

A script may create a new environment variable or change the contents of an existing one with EnvSet. Such additions and changes are not seen by the rest of the system. However, any programs or scripts which the script launches by calling Run or RunWait usually inherit a copy of the parent script's environment variables.

To retrieve an environment variable, use EnvGet. For example:

```
Path := EnvGet("PATH")
```

Variable References (VarRef)

Within an expression, each variable reference is automatically resolved to its contents, unless it is the target of an assignment or the reference operator (&). In other words, calling myFunction(myVar) would pass the value of myVar to myFunction, not the variable itself. The function would then have its own local variable (the parameter) with the same value as myVar, but would not be able to assign a new value to myVar. In short, the parameter is passed by value.

The reference operator (&) allows a variable to be handled like a value. &myVar produces a VarRef, which can be used like any other value: assigned to another variable or property, inserted into an array, passed to or returned from a function, etc. A VarRef can be used to assign to the original target variable or retrieve its value by dereferencing it.

For convenience, a function parameter can be declared ByRef by prefixing the parameter name with ampersand (&). This requires the caller to pass a VarRef, and allows the function itself to "dereference" the VarRef by just referring to the parameter (without percent signs).

class VarRef extends Any

The VarRef class currently has no predefined methods or properties, but value is VarRef can be used to test if a value is a VarRef.

When a VarRef is used as a parameter of a COM method, the object itself is not passed. Instead, its value is copied into a temporary VARIANT, which is passed using the variant type VT_BYREF|VT_VARIANT. When the method returns, the new value is assigned to the VarRef.

Caching

Although a variable is typically thought of as holding a single value, and that value having a distinct type (string, number or object), AutoHotkey automatically converts between numbers and strings in cases like "Value is " myNumber and MsgBox myNumber. As these conversions can happen very frequently, whenever a variable containing a number is converted to a string, the result is cached in the variable.

Currently, AutoHotkey v2 caches a pure number only when assigning a pure number to a variable, not when reading it. This preserves the ability to differentiate between strings and pure numbers (such as with the Type function, or when passing values to COM objects).

Related

Variables: basic usage and examples.

Variable Capacity and Memory: details about limitations.

Functions

A function is the basic means by which a script does something.

Functions can have many different purposes. Some functions might do no more than perform a simple calculation, while others have immediately visible effects, such as moving a window. One of AutoHotkey's strengths is the ease with which scripts can automate other programs and perform many other common tasks by simply calling a few functions. See the function list for examples.

Throughout this documentation, some common words are used in ways that might not be obvious to someone without prior experience. Below are several such words/phrases which are used frequently in relation to functions:

Call a function

Calling a function causes the program to invoke, execute or evaluate it. In other words, a function call temporarily transfers control from the script to the function. When the function has completed its purpose, it returns control to the script. In other words, any code following the function call does not execute until after the function completes.

However, sometimes a function completes before its effects can be seen by the user. For example, the Send function sends keystrokes, but may return before the keystrokes reach their destination and cause their intended effect.

Parameters

Usually a function accepts parameters which tell it how to operate or what to operate on. Each parameter is a value, such as a string or number. For example, WinMove moves a window, so its parameters tell it which window to move and where to move it to. Parameters can also be called arguments. Common abbreviations include param and arg.

Pass parameters

Parameters are passed to a function, meaning that a value is specified for each parameter of the function when it is called. For example, one can pass the name of a key to GetKeyState to determine whether that key is being held down.

Return a value

Functions return a value, so the result of the function is often called a return value. For example, StrLen returns the number of characters in a string. Functions may also store results in variables, such as when there is more than one result (see Returning Values).

Command

A function call is sometimes called a command, such as if it commands the program to take a specific action. (For historical reasons, command may refer to a particular style of calling a function, where the parentheses are omitted and the return value is discarded. However, this is technically a function call statement.)

Functions usually expect parameters to be written in a specific order, so the meaning of each parameter value depends on its position in the comma-delimited list of parameters. Some parameters can be omitted, in which case the parameter can be left blank, but the comma following it can only be omitted if all remaining parameters are also omitted. For example, the syntax for ControlSend is:

ControlSend Keys , Control, WinTitle, WinText, ExcludeTitle, ExcludeText

Square brackets signify that the enclosed parameters are optional (the brackets themselves should not appear in the actual code). However, usually one must also specify the target window. For example:

```
ControlSend "^{Home}", "Edit1", "A"  ; Correct. Control is specified.
ControlSend "^{Home}", "A"          ; Incorrect: Parameters are mismatched.
ControlSend "^{Home}",, "A"         ; Correct. Control is omitted.
```

Methods

A method is a function associated with a specific object or type of object. To call a method, one must specify an object and a method name. The method name does not uniquely identify the function; instead, what happens when the method call is attempted depends on the object. For example, x.Show() might show a menu, show a GUI, raise an error or do something else, depending on what x is. In other words, a method call simply passes a message to the object, instructing it to do something. For details, see Object Protocol and Operators for Objects.

Control Flow

Control flow is the order in which individual statements are executed. Normally statements are executed sequentially from top to bottom, but a control flow statement can override this, such as by specifying that statements should be executed repeatedly, or only if a certain condition is met.

Statement

A statement is simply the smallest standalone element of the language that expresses some action to be carried out. In AutoHotkey, statements include assignments, function calls and other expressions. However, directives, double-colon hotkey and hotstring tags, and declarations without assignments are not statements; they are processed when the program first starts up, before the script executes.

Execute

Carry out, perform, evaluate, put into effect, etc. Execute basically has the same meaning as in non-programming speak.

Body

The body of a control flow statement is the statement or group of statements to which it applies. For example, the body of an if statement is executed only if a specific condition is met.

For example, consider this simple set of instructions:

Open Notepad

Wait for Notepad to appear on the screen

Type "Hello, world!"

We take one step at a time, and when that step is finished, we move on to the next step. In the same way, control in a program or script usually flows from one statement to the next statement. But what if we want to type into an existing Notepad window? Consider this revised set of instructions:

If Notepad is not running:

Open Notepad

Wait for Notepad to appear on the screen

Otherwise:

Activate Notepad

Type "Hello, world!"

So we either open Notepad or activate Notepad depending on whether it is already running. #1 is a conditional statement, also known as an if statement; that is, we execute its body (#1.1 - #1.2) only if a condition is met. #2 is an else statement; we execute its body (#2.1) only if the condition of a previous if statement (#1) is not met. Depending on the condition, control flows one of two ways: #1 (if true) → #1.1 → #1.2 → #3; or #1 (if false) → #2 (else) → #2.1 → #3.

The instructions above can be translated into the code below:

```
if (not WinExist("ahk_class Notepad"))
{
   Run "Notepad"
   WinWait "ahk_class Notepad"
}
else
   WinActivate "ahk_class Notepad"
Send "Hello, world{!}"
```

In our written instructions, we used indentation and numbering to group the statements. Scripts work a little differently. Although indentation makes code easier to read, in AutoHotkey it does not affect the grouping of statements. Instead, statements are grouped by enclosing them in braces, as shown above. This is called a block.

For details about syntax - that is, how to write or recognise control flow statements in AutoHotkey - see Control Flow.

Details

String Encoding

Each character in the string is represented by a number, called its ordinal number, or character code. For example, the value "Abc" would be represented as follows:

A	b	c	
65	98	99	0

Encoding: The encoding of a string defines how symbols are mapped to ordinal numbers, and ordinal numbers to bytes. There are many different encodings, but as all of those supported by AutoHotkey include ASCII as a subset, character codes 0 to 127 always have the same meaning. For example, 'A' always has the character code 65.

Null-termination: Each string is terminated with a "null character", or in other words, a character with ordinal value of zero marks the end of the string. The length of the string can be inferred by the position of the null-terminator, but AutoHotkey also stores the length, for performance and to permit null characters within the string's length.

Note: Due to reliance on null-termination, many built-in functions and most expression operators do not support strings with embedded null characters, and instead read only up to the first null character. However, basic manipulation of such strings is supported; e.g. concatenation, ==, !==, Chr(0), StrLen, SubStr, assignments, parameter values and return.

Native encoding: Although AutoHotkey provides ways to work with text in various encodings, the built-in functions--and to some degree the language itself--all assume string values to be in one particular encoding. This is referred to as the native encoding. The native encoding depends on the version of AutoHotkey:

Unicode versions of AutoHotkey use UTF-16. The smallest element in a UTF-16 string is two bytes (16 bits). Unicode characters in the range 0 to 65535 (U+FFFF) are represented by a single 16-bit code unit of the same value, while characters in the range 65536 (U+10000) to 1114111 (U+10FFFF) are represented by a surrogate pair; that is, exactly two 16-bit code units between 0xD800 and 0xDFFF. (For further explanation of surrogate pairs and methods of encoding or decoding them, search the Internet.)

ANSI versions of AutoHotkey use the system default ANSI code page, which depends on the system locale or "language for non-Unicode programs" system setting. The smallest element of an ANSI string is one byte. However, some code pages contain characters which are represented by sequences of multiple bytes (these are always non-ASCII characters).

Note: AutoHotkey v2 natively uses Unicode and does not have an ANSI version.

Character: Generally, other parts of this documentation use the term "character" to mean a string's smallest unit; bytes for ANSI strings and 16-bit code units for Unicode (UTF-16) strings. For practical reasons, the length of a string and positions within a string are measured by counting these fixed-size units, even though they may not be complete Unicode characters.

FileRead, FileAppend, FileOpen and the File object provide ways of reading and writing text in files with a specific encoding.

The functions StrGet and StrPut can be used to convert strings between the native encoding and some other specified encoding. However, these are usually only useful in combination with data structures and the DllCall function. Strings which are passed directly to or from DllCall can be converted to ANSI or UTF-16 by using the AStr or WStr parameter types.

Techniques for dealing with the differences between ANSI and Unicode versions of AutoHotkey can be found under Unicode vs ANSI.

Pure Numbers

A pure or binary number is one which is stored in memory in a format that the computer's CPU can directly work with, such as to perform math. In most cases, AutoHotkey automatically converts between numeric strings and pure numbers as needed, and rarely differentiates between the two types. AutoHotkey primarily uses two data types for pure numbers:

64-bit signed integers (int64).

64-bit binary floating-point numbers (the double or binary64 format of the IEEE 754 international standard).

In other words, scripts are affected by the following limitations:

Integers must be within the signed 64-bit range; that is, -9223372036854775808 (-0x8000000000000000, or -263) to 9223372036854775807 (0x7FFFFFFFFFFFFFFF, or 263-1). If an integer constant in an expression is outside this range, only the low 64 bits are used (the value is truncated). Although larger values can be contained within a string, any attempt to convert the string to a number (such as by using it in a math operation) will cause it to be similarly truncated.

Floating-point numbers generally support 15 digits of precision.

Note: There are some decimal fractions which the binary floating-point format cannot precisely represent, so a number is rounded to the closest representable number. This may lead to unexpected results. For example:

```
MsgBox 0.1 + 0          ; 0.10000000000000001

MsgBox 0.1 + 0.2        ; 0.30000000000000004

MsgBox 0.3 + 0          ; 0.29999999999999999

MsgBox 0.1 + 0.2 = 0.3  ; 0 (not equal)
```

One strategy for dealing with this is to avoid direct comparison, instead comparing the difference. For example:

```
MsgBox Abs((0.1 + 0.2) - (0.3)) < 0.0000000000000001
```

Another strategy is to explicitly apply rounding before comparison, such as by converting to a string. There are generally two ways to do this while specifying the precision, and both are shown below:

```
MsgBox Round(0.1 + 0.2, 15) = Format("{:.15f}", 0.3)
```

Names

AutoHotkey uses the same set of rules for naming various things, including variables, functions, window groups, classes, properties and methods. The rules are as follows.

Case sensitivity: None for ASCII characters. For example, CurrentDate is the same as currentdate. However, uppercase non-ASCII characters such as 'Ä' are not considered equal to their lowercase counterparts, regardless of the current user's locale. This helps the script to behave consistently across multiple locales.

Maximum length: 253 characters.

Allowed characters: Letters, digits, underscore and non-ASCII characters; however, the first character cannot be a digit.

Reserved words: as, and, contains, false, in, is, IsSet, not, or, super, true, unset. These words are reserved for future use or other specific purposes.

Declaration keywords and names of control flow statements are also reserved, primarily to detect mistakes. This includes: Break, Catch, Continue, Else, Finally, For, Global, Goto, If, Local, Loop, Return, Static, Throw, Try, Until, While

Names of properties, methods and window groups are permitted to be reserved words.

References to Objects

Scripts interact with an object only indirectly, through a reference to the object. When you create an object, the object is created at some location you don't control, and you're given a reference. Passing this reference to a function or storing it in a variable or another object creates a new reference to the same object.

For example, if myObj contains a reference to an object, yourObj := myObj creates a new reference to the same object. A change such as myObj.ans := 42 would be reflected by both myObj.ans and yourObj.ans, since they both refer to the same object. However, myObj := Object() only affects the variable myObj, not the variable yourObj, which still refers to the original object.

A reference is released by simply using an assignment to replace it with any other value. An object is deleted only after all references have been released; you cannot delete an object explicitly, and should not try. (However, you can delete an object's properties, content or associated resources, such as an Array's elements, the window associated with a Gui, the menu of a Menu object, and so on.)

```
ref1 := Object()  ; Create an object and store first reference
ref2 := ref1      ; Create a new reference to the same object
ref1 := ""        ; Release the first reference
ref2 := ""        ; Release the second reference; object is deleted
```

If that's difficult to understand, try thinking of an object as a rental unit. When you rent a unit, you're given a key which you can use to access the unit. You can get more keys and use them to access the same unit, but when you're finished with the unit, you must hand all keys back to the rental agent. Usually a unit wouldn't be deleted, but maybe the agent will have any junk you left behind removed; just as any values you stored in an object are freed when the object is deleted.

Scripting Language

An AutoHotkey script is basically a set of instructions for the program to follow, written in a custom language exclusive to AutoHotkey. This language bears some similarities to several other scripting languages, but also has its own unique strengths and pitfalls. This document describes the language and also tries to point out common pitfalls.

See Concepts and Conventions for more general explanation of various concepts utilised by AutoHotkey.

Table of Contents

Expression Statements

Control Flow Statements

Control Flow vs. Other Statements

Loop Statement

Not Control Flow

Structure of a Script

Global Code

Functions

#Include

Miscellaneous

Dynamic Variables

Pseudo-arrays

Labels

General Conventions

Names: Variable and function names are not case sensitive (for example, CurrentDate is the same as currentdate). For details such as maximum length and usable characters, see Names.

No typed variables: Variables have no explicitly defined type; instead, a value of any type can be stored in any variable (excluding constants and built-in variables). Numbers may be automatically converted to strings (text) and vice versa, depending on the situation.

Declarations are optional: Except where noted on the functions page, variables do not need to be declared. However, attempting to read a variable before it is given a value is considered an error.

Spaces are mostly ignored: Indentation (leading space) is important for writing readable code, but is not required by the program and is generally ignored. Spaces and tabs are generally ignored at the end of a line and within an expression (except between quotes). However, spaces are significant in some cases, including:

Function and method calls require there to be no space between the function/method name and (.

Spaces are required when performing concatenation.

Spaces may be required between two operators, to remove ambiguity.

Single-line comments require a leading space if they are not at the start of the line.

Line breaks are meaningful: Line breaks generally act as a statement separator, terminating the previous function call or other statement. (A statement is simply the smallest standalone element of the language that expresses some action to be carried out.) The exception to this is line continuation (see below).

Line continuation: Long lines can be divided up into a collection of smaller ones to improve readability and maintainability. This is achieved by preprocessing, so is not part of the language as such. There are three methods:

Continuation prefix: Lines that begin with an expression operator (except ++ and --) are merged with the previous line. Lines are merged regardless of whether the line actually contains an expression.

Continuation by enclosure: A sub-expression enclosed in (), [] or {} can automatically span multiple lines in most cases.

Continuation section: Multiple lines are merged with the line above the section, which starts with (and ends with) (both symbols must appear at the beginning of a line, excluding whitespace).

Comments

Comments are portions of text within the script which are ignored by the program. They are typically used to add explanation or disable parts of the code.

Scripts can be commented by using a semicolon at the beginning of a line. For example:

```
; This entire line is a comment.
```

Comments may also be added at the end of a line, in which case the semicolon must have at least one space or tab to its left. For example:

```
Run "Notepad"  ; This is a comment on the same line as a function call.
```

In addition, the /* and */ symbols can be used to comment out an entire section, as in this example:

```
/*
MsgBox "This line is commented out (disabled)."
MsgBox "Common mistake:" */ " this does not end the comment."
MsgBox "This line is commented out."
*/
MsgBox "This line is not commented out."
/* This is also valid, but no other code can share the line. */
MsgBox "This line is not commented out."
```

Excluding tabs and spaces, /* must appear at the start of the line, while */ can appear only at the start or end of a line. It is also valid to omit */, in which case the remainder of the file is commented out.

Since comments are filtered out when the script is read from file, they do not impact performance or memory utilization.

Expressions

Expressions are combinations of one or more values, variables, operators and function calls. For example, 10, 1+1 and MyVar are valid expressions. Usually, an expression takes one or more values as input, performs one or more operations, and produces a value as the result. The process of finding out the value of an expression is called evaluation. For example, the expression 1+1 evaluates to the number 2.

Simple expressions can be pieced together to form increasingly more complex expressions. For example, if Discount/100 converts a discount percentage to a fraction, 1 - Discount/100 calculates a fraction representing the remaining amount, and Price * (1 - Discount/100) applies it to produce the net price.

Values are numbers, objects or strings. A literal value is one written physically in the script; one that you can see when you look at the code.

Strings / Text

For a more general explanation of strings, see Strings.

A string, or string of characters, is just a text value. In an expression, literal text must be enclosed in single or double quotation marks to differentiate it from a variable name or some other expression. This is often referred to as a quoted literal string, or just quoted string. For example, "this is a quoted string" and 'so is this'.

To include an actual quote character inside a quoted string, use the `" or `' escape sequence or enclose the character in the opposite type of quote mark. For example: 'She said, "An apple a day."'.

Quoted strings can contain other escape sequences such as `t (tab), `n (linefeed), and `r (carriage return).

Variables

For a basic explanation and general details about variables, see Variables.

Variables can be used in an expression simply by writing the variable's name. For example, A_ScreenWidth/2. However, variables cannot be used inside a quoted string. Instead, variables and other values can be combined with text through a process called concatenation. There are two ways to concatenate values in an expression:

Implicit concatenation: "The value is " MyVar

Explicit concatenation: "The value is " . MyVar

Implicit concatenation is also known as auto-concat. In both cases, the spaces preceding the variable and dot are mandatory.

The Format function can also be used for this purpose. For example:

MsgBox Format("You are using AutoHotkey v{1} {2}-bit.", A_AhkVersion, A_PtrSize*8)

To assign a value to a variable, use the := assignment operator, as in MyVar := "Some text".

Percent signs within an expression are used to create dynamic variable references, but these are rarely needed.

Keyword Constants

A constant is simply an unchangeable value, given a symbolic name. AutoHotkey currently has the following constants:

Name	Value	Type	Description
False	0	Integer	Boolean false, sometimes meaning "off", "no", etc.
True	1	Integer	Boolean true, sometimes meaning "on", "yes", etc.

Unlike the read-only built-in variables, these cannot be returned by a dynamic reference.

Operators

Operators take the form of a symbol or group of symbols such as + or :=, or one of the words and, or, not, is, in or contains. They take one, two or three values as input and return a value as the result. A value or sub-expression used as input for an operator is called an operand.

Unary operators are written either before or after a single operand, depending on the operator. For example, -x or not keyIsDown.

Binary operators are written in between their two operands. For example, 1+1 or 2 * 5.

AutoHotkey has only one ternary operator, which takes the form condition ? valueIfTrue : valueIfFalse.

Some unary and binary operators share the same symbols, in which case the meaning of the operator depends on whether it is written before, after or in between two values. For example, x-y performs subtraction while -x inverts the sign of x (producing a positive value from a negative value and vice versa).

Operators of equal precedence such as multiply (*) and divide (/) are evaluated in left-to-right order unless otherwise specified in the operator table. By contrast, an operator of lower precedence such as add (+) is evaluated after a higher one such as multiply (*). For example, 3 + 2 * 2 is evaluated as 3 + (2 * 2). Parentheses may be used to override precedence as in this example: (3 + 2) * 2

Function Calls

For a general explanation of functions and related terminology, see Functions.

Functions take a varying number of inputs, perform some action or calculation, and then return a result. The inputs of a function are called parameters or arguments. A function is called simply by writing the target function followed by parameters enclosed in parentheses. For example, GetKeyState("Shift") returns (evaluates to) 1 if Shift is being held down or 0 otherwise.

Note: There must not be any space between the function and open parenthesis.

For those new to programming, the requirement for parentheses may seem cryptic or verbose at first, but they are what allows a function call to be combined with other operations. For example, the expression GetKeyState("Shift", "P") and GetKeyState("Ctrl", "P") returns 1 only if both keys are being physically held down.

Although a function call expression usually begins with a literal function name, the target of the call can be any expression which produces a function object. In the expression GetKeyState("Shift"), GetKeyState is actually a variable reference, although it usually refers to a read-only variable containing a built-in function.

Function Call Statements

If the return value of the function is not needed and the function name is written at the start of the line (or in other contexts which allow a statement, such as following else or a hotkey), the parentheses can be omitted. In this case, the remainder of the line is taken as the function's parameter list. For example:

result := MsgBox("This one requires parentheses.",, "OKCancel")

MsgBox "This one doesn't. The result was " result "."

Parentheses can also be omitted when calling a method in this same context, but only when the target object is either a variable or a directly named property, such as myVar.myMethod or myVar.myProp.myMethod.

As with function call expressions, the target of a function call statement does not have to be a predefined function; it can instead be a variable containing a function object.

A function call statement can span multiple lines.

Function call statements have the following limitations:

If there is a return value, it is always discarded.

Like control flow statements, they cannot be used inside an expression.

When optional parameters are omitted, any commas at the end of the parameter list must also be omitted to prevent line continuation.

Function call statements cannot be variadic, although they can pass a fixed number of parameters to a variadic function.

Optional Parameters

Optional parameters can simply be left blank, but the delimiting comma is still required unless all subsequent parameters are also omitted. For example, the Run function can accept between one and four parameters. All of the following are valid:

Run "notepad.exe", "C:\"

Run "notepad.exe",, "Min"

Run("notepad.exe", , , &notepadPID)

Within a function call, array literal or object literal, the keyword unset can be used to explicitly omit the parameter or value. An unset expression has one of the following effects:

For a user-defined function, the parameter's default value is used.

For a built-in function, the parameter is considered to have been omitted.

For an array literal such as [var?], the element is included in the array's length but is given no value.

For an object literal such as {x: y?}, the property is not assigned.

The unset keyword can also be used in a function definition to indicate that a parameter is optional but has no default value. When the function executes, the local variable corresponding to that parameter will have no value if the parameter was omitted.

The maybe operator (var?) can be used to pass or omit a variable depending on whether it has a value. For example, Array(MyVar?) is equivalent to Array(IsSet(MyVar) ? MyVar : unset).

Operators for Objects

There are other symbols used in expressions which don't quite fit into any of the categories defined above, or that affect the meaning of other parts of the expression, as described below. These all relate to objects in some way. Providing a full explanation of what each construct does would require introducing more concepts which are outside the scope of this section.

Alpha.Beta is often called member access. Alpha is an ordinary variable, and could be replaced with a function call or some other sub-expression which returns an object. When evaluated, the object is sent a request "give me the value of property Beta", "store this value in property Beta" or "call the method named Beta". In other words, Beta is a name which has meaning to the object; it is not a local or global variable.

Alpha.Beta() is a method call, as described above. The parentheses can be omitted in specific cases; see Function Call Statements.

Alpha.Beta[Param] is a specialised form of member access which includes additional parameters in the request. While Beta is a simple name, Param is an ordinary variable or sub-expression, or a list of sub-expressions separated by commas (the same as in a function's parameter list). Variadic calls are permitted.

Alpha.%vBeta%, Alpha.%vBeta%[Param] and Alpha.%vBeta%() are also member access, but vBeta is a variable or sub-expression. This allows the name of the property or method to be determined while the script is running. Parentheses are required when calling a method this way.

Alpha[Index] accesses the default property of Alpha, giving Index as a parameter. Both Alpha and Index are variables in this case, and could be replaced with virtually any sub-expression. This syntax is usually used to retrieve an element of an Array or Map.

[A, B, C] creates an Array with the initial contents A, B and C (all variables in this case), where A is element 1.

{Prop1: Value1, Prop2: Value2} creates an Object with properties literally named Prop1 and Prop2. A value can later be retrieved by using the member access syntax described above. To evaluate a property name as an expression, enclose it in percent signs. For example: {%NameVar%: ValueVar}.

MyFunc(Params*) is a variadic function call. The asterisk must immediately precede the closing parenthesis at the end of the function's parameter list. Params must be a variable or sub-expression which returns an Array or other enumerable object. Although it isn't valid to use Params* just anywhere, it can be used in an array literal ([A, B, C, ArrayToAppend*]) or property parameter list (Alpha.Beta[Params*] or Alpha[Params*]).

Expression Statements

Not all expressions can be used alone on a line. For example, a line consisting of just 21*2 or "Some text" wouldn't make any sense. An expression statement is an expression used on its own, typically for its side-effects. Most expressions with side-effects can be used this way, so it is generally not necessary to memorise the details of this section.

The following types of expressions can be used as statements:

Assignments, as in x := y, compound assignments such as x += y, and increment/decrement operators such as ++x and x--.

Known limitation: For x++ and x--, there currently cannot be a space between the variable name and operator.

Function calls such as MyFunc(Params). However, a standalone function call cannot be followed by an open brace { (at the end of the line or on the next line), because it would be confused with a function declaration.

Method calls such as MyObj.MyMethod().

Member access using square brackets, such as MyObj[Index], which can have side-effects like a function call.

Ternary expressions such as x ? CallIfTrue() : CallIfFalse(). However, it is safer to utilize the rule below; that is, always enclose the expression (or just the condition) in parentheses.

Known limitation: Due to ambiguity with function call statements, conditions beginning with a variable name and space (but also containing other operators) should be enclosed in parentheses. For example, (x + 1) ? y : z and x+1 ? y : z are expression statements but x + 1 ? y : z is a function call statement.

Note: The condition cannot begin with ! or any other expression operator, as it would be interpreted as a continuation line.

Expressions starting with (. However, there usually must be a matching) on the same line, otherwise the line would be interpreted as the start of a continuation section.

Expressions starting with a double-deref, such as %varname% := 1. This is primarily due to implementation complexity.

Expressions that start with any of those described above (but not those described below) are also allowed, for simplicity. For example, MyFunc()+1 is currently allowed, although the +1 has no effect and its result is discarded. Such expressions might become invalid in the future due to enhanced error-checking.

Function call statements are similar to expression statements, but are technically not pure expressions. For example, MsgBox "Hello, world!", myGui.Show or x.y.z "my parameter".

Control Flow Statements

For a general explanation of control flow, see Control Flow.

Statements are grouped together into a block by enclosing them in braces {}, as in C, JavaScript and similar languages, but usually the braces must appear at the start of a line. Control flow statements can be applied to an entire block or just a single statement.

The body of a control flow statement is always a single group of statements. A block counts as a single group of statements, as does a control flow statement and its body. The following related statements are also grouped with each other, along with their bodies: If with Else; Loop/For with Until or Else; Try with Catch and/or Else and/or Finally. In other words, when a group of these statements is used as a whole, it does not always need to be enclosed in braces (however, some coding styles always include the braces, for clarity).

Control flow statements which have a body and therefore must always be followed by a related statement or group of statements: If, Else, Loop, While, For, Try, Catch and Finally.

The following control flow statements exist:

A block (denoted by a pair of braces) groups zero or more statements to act as a single statement.

An If statement causes its body to be executed or not depending on a condition. It can be followed by an Else statement, which executes only if the condition was not met.

Goto jumps to the specified label and continues execution.

Return returns from a function.

A Loop statement (Loop, While or For) executes its body repeatedly.

Break exits (terminates) a loop.

Continue skips the rest of the current loop iteration and begins a new one.

Until causes a loop to terminate when an expression evaluates to true. The expression is evaluated after each iteration.

Switch executes one case from a list of mutually exclusive candidates.

Exception handling:

Try guards its body against runtime errors and values thrown by the throw statement.

Catch executes if an exception of a given type is thrown within a try statement.

Else, when used after a catch statement, executes only if no exception is thrown within a try statement.

Finally executes its body when control is being transferred out of a try or catch statement's body.

Throw throws an exception to be handled by try/catch or OnError, or to display an error dialog.

Control Flow vs. Other Statements

Control flow statements differ from function call statements in several ways:

The opening brace of a block can be written at the end of the same line as an If, Else, Loop, While, For, Try, Catch or Finally statement (basically any control flow statement which has a body). This is referred to as the One True Brace (OTB) style.

Else, Try and Finally allow any valid statement to their right, as they require a body but have no parameters.

If, While, Return, Until, Loop Count and Goto allow an open parenthesis to be used immediately after the name, to enclose the entire parameter list. Although these look like function calls, they are not, and cannot be used mid-expression. For example, if(expression).

Control flow statements cannot be overridden by defining a function with the same name.

Loop Statement

There are several types of loop statements:

Loop Count executes a statement repeatedly: either the specified number of times or until break is encountered.

Loop Reg retrieves the contents of the specified registry subkey, one item at a time.

Loop Files retrieves the specified files or folders, one at a time.

Loop Parse retrieves substrings (fields) from a string, one at a time.

Loop Read retrieves the lines in a text file, one at a time.

While executes a statement repeatedly until the specified expression evaluates to false. The expression is evaluated before each iteration.

For executes a statement once for each value or pair of values returned by an enumerator, such as each key-value pair in an object.

Break exits (terminates) a loop, effectively jumping to the next line after the loop's body.

Continue skips the rest of the current loop iteration and begins a new one.

Until causes a loop to terminate when an expression evaluates to true. The expression is evaluated after each iteration.

A label can be used to "name" a loop for Continue and Break. This allows the script to easily continue or break out of any number of nested loops without using Goto.

The built-in variable A_Index contains the number of the current loop iteration. It contains 1 the first time the loop's body is executed. For the second time, it contains 2; and so on. If an inner loop is enclosed by an outer loop, the inner loop takes precedence. A_Index works inside all types of loops, but contains 0 outside of a loop.

For some loop types, other built-in variables return information about the current loop item (registry key/value, file, substring or line of text). These variables have names beginning with A_Loop, such as A_LoopFileName and A_LoopReadLine. Their values always correspond to the most recently started (but

not yet stopped) loop of the appropriate type. For example, A_LoopField returns the current substring in the innermost parsing loop, even if it is used inside a file or registry loop.

```
t := "column 1`tcolumn 2`nvalue 1`tvalue 2"
Loop Parse t, "`n"
{
   rowtext := A_LoopField
   rownum := A_Index  ; Save this for use in the second loop, below.
   Loop Parse rowtext, "`t"
   {
      MsgBox rownum ":" A_Index " = " A_LoopField
   }
}
```

Loop variables can also be used outside the body of a loop, such as in a function which is called from within a loop.

Not Control Flow

As directives, labels, double-colon hotkey and hotstring tags, and declarations without assignments are processed when the script is loaded from file, they are not subject to control flow. In other words, they take effect unconditionally, before the script ever executes any control flow statements. Similarly, the #HotIf directive cannot affect control flow; it merely sets the criteria for any hotkeys and hotstrings specified in the code. A hotkey's criteria is evaluated each time it is pressed, not when the #HotIf directive is encountered in the code.

Structure of a Script

Global Code

After the script has been loaded, the auto-execute thread begins executing at the script's top line, and continues until instructed to stop, such as by Return, ExitApp or Exit. The physical end of the script also acts as Exit.

Global code, or code in the global scope, is any executable code that is not inside a function or class definition. Any variable references there are said to be global, since they can be accessed by any function (with the proper declaration). Such code is often used to configure settings which apply to every newly launched thread, or to initialize global variables used by hotkeys and other functions.

Code to be executed at startup (immediately when the script starts) is often placed at the top of the file. However, such code can be placed throughout the file, in between (but not inside) function and class definitions. This is because the body of each function or class definition is skipped whenever it is encountered during execution. In some cases, the entire purpose of the script may be carried out with global code.

Related: Script Startup (the Auto-execute Thread)

Subroutines

A subroutine (also sub or procedure) is a reusable block of code which can be executed on demand. A subroutine is created by defining a function (see below). These terms are generally interchangeable for AutoHotkey v2, where functions are the only type of subroutine.

Functions

Related: Functions (all about defining functions)

Aside from calling the many useful predefined functions, a script can define its own functions. These functions can generally be used two ways:

A function can be called by the script itself. This kind of function might be used to avoid repetition, to make the code more manageable, or perhaps for other purposes.

A function can be called by the program in response to some event, such as the user pressing a hotkey. For instance, each hotkey is associated with a function to execute whenever the hotkey is pressed.

There are multiple ways to define a function:

A function definition combining a name, parentheses and a block of code. This defines a function which can be executed by name with a function call or function call statement. For example:

```
SayHello()  ; Define the SayHello function.
{
   MsgBox "Hello!"
}

SayHello  ; Call the SayHello function.
```

A hotkey or hotstring definition, combining a hotkey or hotstring with a single statement or a block of code. This type of function cannot be called directly, but is executed whenever the hotkey or hotstring is activated. For example:

```
#w::Run "wordpad"  ; Press Win-W to run Wordpad.

#n::  ; Press Win-N to run Notepad.
{
   Run "notepad"
}
```

A fat arrow expression defines a function which evaluates an expression and returns its result, instead of executing a block of code. Such functions usually have no name as they are passed directly to another function. For example:

```
SetTimer () => MsgBox("Hello!"), -1000  ; Says hello after 1 second.
```

The fat arrow syntax can also be used outside of expressions as shorthand for a normal function or method definition. For example, the following is equivalent to the SayHello definition above, except that this one returns "OK":

```
SayHello() => MsgBox("Hello!")
```

Variables in functions are local to that function by default, except in the following cases:

When the function is assume-global.

When a variable is referenced but not used as the target of an assignment or the reference operator (&var).

When referring to a local variable of an outer function inside a nested function.

A function can optionally accept parameters. Parameters are defined by listing them inside the parentheses. For example:

```
MyFunction(FirstParameter, Second, &Third, Fourth:="")
{
   ;…
   return "a value"
}
```

As with function calls, there must be no space between the function name and open-parenthesis.

The line break between the close-parenthesis and open-brace is optional. There can be any amount of whitespace or comments between the two.

The ByRef marker (&) indicates that the caller must pass a variable reference. Inside the function, any reference to the parameter will actually access the caller's variable. This is similar to omitting & and explicitly dereferencing the parameter inside the function (e.g. %Third%), but in this case the percent signs are omitted. If the parameter is optional and the caller omits it, the parameter acts as a normal local variable.

Optional parameters are specified by following the parameter name with := and a default value, which must be a literal quoted string, a number, true, false or unset.

The function can return a value. If it does not, the default return value is an empty string.

A function definition does not need to precede calls to that function.

See Functions for much more detail.

#Include

The #Include directive causes the script to behave as though the specified file's contents are present at this exact position. This is often used to organise code into separate files, or to make use of script libraries written by other users.

An #Include file can contain global code to be executed during script startup, but as with code in the main script file, such code will be executed only if the auto-execute thread is not terminated (such as with an unconditional Return) prior to the #Include directive. A warning is displayed by default if any code cannot be executed due to a prior Return.

Unlike in C/C++, #Include does nothing if the file has already been included by a previous directive. To include the contents of the same file multiple times, use #IncludeAgain.

To facilitate sharing scripts, #Include can search a few standard locations for a library script. For details, see Script Library Folders.

Miscellaneous

Dynamic Variables

A dynamic variable reference takes a text value and interprets it as the name of a variable.

Note: A variable cannot be created by a dynamic reference, but existing variables can be assigned. This includes all variables which the script contains non-dynamic references to, even if they have not been assigned values.

The most common form of dynamic variable reference is called a double reference or double-deref. Before performing a double reference, the name of the target variable is stored in a second variable. This second variable can then be used to assign a value to the target variable indirectly, using a double reference. For example:

```
target := 42

second := "target"

MsgBox  second   ; Normal (single) variable reference => target

MsgBox %second%  ; Double-deref => 42
```

Currently, second must always contain a variable name in the second case; arbitrary expressions are not supported.

A dynamic variable reference can also take one or more pieces of literal text and the content of one or more variables, and join them together to form a single variable name. This is done simply by writing the pieces of the name and percent-enclosed variables in sequence, without any spaces. For example, MyArray%A_Index% or MyGrid%X%_%Y%. This is used to access pseudo-arrays, described below.

These techniques can also be applied to properties and methods of objects. For example:

```
clr := {}

for n, component in ["red", "green", "blue"]

   clr.%component% := Random(0, 255)

MsgBox clr.red "," clr.green "," clr.blue
```

Pseudo-arrays

A pseudo-array is actually just a bunch of discrete variables, but with a naming pattern which allows them to be used like elements of an array. For example:

```
MyArray1 := "A"

MyArray2 := "B"

MyArray3 := "C"

Loop 3
```

```
MsgBox MyArray%A_Index%  ; Shows A, then B, then C.
```

The "index" used to form the final variable name does not have to be numeric; it could instead be a letter or keyword.

For these reasons, it is generally recommended to use an Array or Map instead of a pseudo-array:

As the individual elements are just normal variables, one can assign or retrieve a value, but cannot remove or insert elements.

Because the pseudo-array is only conceptual and not a single value, it can't be passed to or returned from a function, or copied as a whole.

A pseudo-array cannot be declared as a whole, so some "elements" may resolve to global (or captured) variables while others do not.

If a variable is referenced non-dynamically but only assigned dynamically, a load-time warning may be displayed. Such warnings are a very effective way to detect errors, so disabling them is not recommended.

Current versions of the language do not permit creating new variables dynamically. This is partly to encourage best practices, and partly to avoid inconsistency between dynamic and non-dynamic variable references in functions.

Labels

A label identifies a line of code, and can be used as a Goto target or to specify a loop to break out of or continue. A label consist of a name followed by a colon:

```
this_is_a_label:
```

Aside from whitespace and comments, no other code can be written on the same line as a label. For more details, see Labels.

Hotkeys (Mouse, Joystick and Keyboard Shortcuts)

Table of Contents

Introduction and Simple Examples

Hotkeys are sometimes referred to as shortcut keys because of their ability to easily trigger an action (such as launching a program or keyboard macro). In the following example, the hotkey Win+N is configured to launch Notepad. The pound sign [#] stands for Win, which is known as a modifier key:

```
#n::
{
   Run "notepad"
}
```

In the above, the braces serve to define a function body for the hotkey. The opening brace may also be specified on the same line as the double-colon to support the OTB (One True Brace) style. However, if a hotkey needs to execute only a single line, that line can be listed to the right of the double-colon. In other words, the braces are implicit:

```
#n::Run "notepad"
```

When a hotkey is triggered, the name of the hotkey is passed as its first parameter named ThisHotkey (which excludes the trailing colons). For example:

#n::MsgBox ThisHotkey ; Reports #n

With few exceptions, this is similar to the built-in variable A_ThisHotkey. The parameter name can be changed by using a named function.

To use more than one modifier with a hotkey, list them consecutively (the order does not matter). The following example uses ^!s to indicate Ctrl+Alt+S:

```
^!s::
{
   Send "Sincerely,{enter}John Smith"  ; This line sends keystrokes to the active (foremost) window.
}
```

Hotkey Modifier Symbols

You can use the following modifier symbols to define hotkeys:

Symbol Description

#

Win (Windows logo key).

Hotkeys that include Win (e.g. #a) will wait for Win to be released before sending any text containing an L keystroke. This prevents usages of Send within such a hotkey from locking the PC. This behavior applies to all sending modes except SendPlay (which doesn't need it), blind mode and text mode.

Note: Pressing a hotkey which includes Win may result in extra simulated keystrokes (Ctrl by default). See A_MenuMaskKey.

!

Alt

Note: Pressing a hotkey which includes Alt may result in extra simulated keystrokes (Ctrl by default). See A_MenuMaskKey.

^ Ctrl

+ Shift

& An ampersand may be used between any two keys or mouse buttons to combine them into a custom hotkey. See below for details.

< Use the left key of the pair. e.g. <!a is the same as !a except that only the left Alt will trigger it.

> Use the right key of the pair.

<^>!

AltGr (alternate graph, or alternate graphic). If your keyboard layout has AltGr instead of a right Alt key, this series of symbols can usually be used to stand for AltGr. For example:

<^>!m::MsgBox "You pressed AltGr+m."

<^<!m::MsgBox "You pressed LeftControl+LeftAlt+m."

Alternatively, to make AltGr itself into a hotkey, use the following hotkey (without any hotkeys like the above present):

LControl & RAlt::MsgBox "You pressed AltGr itself."

*

Wildcard: Fire the hotkey even if extra modifiers are being held down. This is often used in conjunction with remapping keys or buttons. For example:

*#c::Run "calc.exe" ; Win+C, Shift+Win+C, Ctrl+Win+C, etc. will all trigger this hotkey.

*ScrollLock::Run "notepad" ; Pressing ScrollLock will trigger this hotkey even when modifier key(s) are down.

Wildcard hotkeys always use the keyboard hook, as do any hotkeys eclipsed by a wildcard hotkey. For example, the presence of *a:: would cause ^a:: to always use the hook.

~

When the hotkey fires, its key's native function will not be blocked (hidden from the system). In both of the below examples, the user's click of the mouse button will be sent to the active window:

```
~RButton::MsgBox "You clicked the right mouse button."

~RButton & C::MsgBox "You pressed C while holding down the right mouse button."
```

Unlike the other prefix symbols, the tilde prefix is allowed to be present on some of a hotkey's variants but absent on others. However, if a tilde is applied to the prefix key of any custom combination which has not been turned off or suspended, it affects the behavior of that prefix key for all combinations.

Special hotkeys that are substitutes for alt-tab always ignore the tilde prefix.

If the tilde prefix is applied to a custom modifier key (prefix key) which is also used as its own hotkey, that hotkey will fire when the key is pressed instead of being delayed until the key is released. For example, the ~RButton hotkey above is fired as soon as the button is pressed.

If the tilde prefix is applied only to the custom combination and not the non-combination hotkey, the key's native function will still be blocked. For example, in the script below, holding Menu will show the tooltip and will not trigger a context menu:

```
AppsKey::ToolTip "Press < or > to cycle through windows."

AppsKey Up::ToolTip

~AppsKey & <::Send "!+{Esc}"

~AppsKey & >::Send "!{Esc}"
```

If at least one variant of a keyboard hotkey has the tilde modifier, that hotkey always uses the keyboard hook.

$

This is usually only necessary if the script uses the Send function to send the keys that comprise the hotkey itself, which might otherwise cause it to trigger itself. The $ prefix forces the keyboard hook to be used to implement this hotkey, which as a side-effect prevents the Send function from triggering it. The $ prefix is equivalent to having specified #UseHook somewhere above the definition of this hotkey.

The $ prefix has no effect for mouse hotkeys, since they always use the mouse hook. It also has no effect for hotkeys which already require the keyboard hook, including any keyboard hotkeys with the tilde (~) or wildcard (*) modifiers, key-up hotkeys and custom combinations. To determine whether a particular hotkey uses the keyboard hook, use ListHotkeys.

#InputLevel and SendLevel provide additional control over which hotkeys and hotstrings are triggered by the Send function.

UP

The word UP may follow the name of a hotkey to cause the hotkey to fire upon release of the key rather than when the key is pressed down. The following example remaps the left Win to become the left Ctrl:

*LWin::Send "{LControl down}"

*LWin Up::Send "{LControl up}"

"Up" can also be used with normal hotkeys as in this example: ^!r Up::MsgBox "You pressed and released Ctrl+Alt+R". It also works with combination hotkeys (e.g. F1 & e Up::)

Limitations: 1) "Up" does not work with joystick buttons; and 2) An "Up" hotkey without a normal/down counterpart hotkey will completely take over that key to prevent it from getting stuck down. One way to prevent this is to add a tilde prefix (e.g. ~LControl up::)

"Up" hotkeys and their key-down counterparts (if any) always use the keyboard hook.

On a related note, a technique similar to the above is to make a hotkey into a prefix key. The advantage is that although the hotkey will fire upon release, it will do so only if you did not press any other key while it was held down. For example:

LControl & F1::return ; Make left-control a prefix by using it in front of "&" at least once.

LControl::MsgBox "You released LControl without having used it to modify any other key."

Note: See the Key List for a complete list of keyboard keys and mouse/joystick buttons.

Multiple hotkeys can be stacked vertically to have them perform the same action. For example:

```
^Numpad0::
^Numpad1::
{
    MsgBox "Pressing either Ctrl+Numpad0 or Ctrl+Numpad1 will display this."
}
```

A key or key-combination can be disabled for the entire system by having it do nothing. The following example disables the right-side Win:

```
RWin::return
```

Context-sensitive Hotkeys

The #HotIf directive can be used to make a hotkey perform a different action (or none at all) depending on a specific condition. For example:

```
#HotIf WinActive("ahk_class Notepad")
^a::MsgBox "You pressed Ctrl-A while Notepad is active. Pressing Ctrl-A in any other window will pass the Ctrl-A keystroke to that window."
#c::MsgBox "You pressed Win-C while Notepad is active."

#HotIf
#c::MsgBox "You pressed Win-C while any window except Notepad is active."
```

#HotIf MouseIsOver("ahk_class Shell_TrayWnd") ; For MouseIsOver, see #HotIf example 1.

WheelUp::Send "{Volume_Up}" ; Wheel over taskbar: increase/decrease volume.

WheelDown::Send "{Volume_Down}" ;

Custom Combinations

Normally shortcut key combinations consist of optional prefix/modifier keys (Ctrl, Alt, Shift and LWin/RWin) and a single suffix key. The standard modifier keys are designed to be used in this manner, so normally have no immediate effect when pressed down.

A custom combination of two keys (including mouse but not joystick buttons) can be defined by using " & " between them. Because they are intended for use with prefix keys that are not normally used as such, custom combinations have the following special behavior:

The prefix key loses its native function, unless it is a standard modifier key or toggleable key such as CapsLock.

If the prefix key is also used as a suffix in another hotkey, by default that hotkey is fired upon release, and is not fired at all if it was used to activate a custom combination. If there is both a key-down hotkey and a key-up hotkey, both hotkeys are fired at once. The fire-on-release effect is disabled if the tilde prefix is applied to the prefix key in at least one active custom combination or the suffix hotkey itself.

Note: For combinations with standard modifier keys, it is usually better to use the standard syntax. For example, use <+s:: rather than LShift & s::.

In the below example, you would hold down Numpad0 then press the second key to trigger the hotkey:

Numpad0 & Numpad1::MsgBox "You pressed Numpad1 while holding down Numpad0."

Numpad0 & Numpad2::Run "Notepad"

The prefix key loses its native function: In the above example, Numpad0 becomes a prefix key; but this also causes Numpad0 to lose its original/native function when it is pressed by itself. To avoid this, a script may configure Numpad0 to perform a new action such as one of the following:

Numpad0::WinMaximize "A" ; Maximize the active/foreground window.

Numpad0::Send "{Numpad0}" ; Make the release of Numpad0 produce a Numpad0 keystroke. See comment below.

Fire on release: The presence of one of the above custom combination hotkeys causes the release of Numpad0 to perform the indicated action, but only if you did not press any other keys while Numpad0 was being held down. This behaviour can be avoided by applying the tilde prefix to either hotkey.

Modifiers: Unlike a normal hotkey, custom combinations act as though they have the wildcard (*) modifier by default. For example, 1 & 2:: will activate even if Ctrl or Alt is held down when 1 and 2 are pressed, whereas ^1:: would be activated only by Ctrl+1 and not Ctrl+Alt+1.

Combinations of three or more keys are not supported. Combinations which your keyboard hardware supports can usually be detected by using #HotIf and GetKeyState, but the results may be inconsistent. For example:

```
; Press AppsKey and Alt in any order, then slash (/).

#HotIf GetKeyState("AppsKey", "P")

Alt & /::MsgBox "Hotkey activated."


; If the keys are swapped, Alt must be pressed first (use one at a time):

#HotIf GetKeyState("Alt", "P")

AppsKey & /::MsgBox "Hotkey activated."


; [ & ] & \::

#HotIf GetKeyState("[") && GetKeyState("]")

\::MsgBox
```

Keyboard hook: Custom combinations involving keyboard keys always use the keyboard hook, as do any hotkeys which use the prefix key as a suffix. For example, a & b:: causes ^a:: to always use the hook.

Other Features

NumLock, CapsLock, and ScrollLock: These keys may be forced to be "AlwaysOn" or "AlwaysOff". For example: SetNumLockState "AlwaysOn".

Overriding Explorer's hotkeys: Windows' built-in hotkeys such as Win+E (#e) and Win+R (#r) can be individually overridden simply by assigning them to an action in the script. See the override page for details.

Substitutes for Alt-Tab: Hotkeys can provide an alternate means of alt-tabbing. For example, the following two hotkeys allow you to alt-tab with your right hand:

```
RControl & RShift::AltTab  ; Hold down right-control then press right-shift repeatedly to move forward.

RControl & Enter::ShiftAltTab  ; Without even having to release right-control, press Enter to reverse direction.
```

For more details, see Alt-Tab.

Mouse Wheel Hotkeys

Hotkeys that fire upon turning the mouse wheel are supported via the key names WheelDown and WheelUp. Here are some examples of mouse wheel hotkeys:

```
MButton & WheelDown::MsgBox "You turned the mouse wheel down while holding down the middle button."

^!WheelUp::MsgBox "You rotated the wheel up while holding down Control+Alt."
```

If the mouse supports it, horizontal scrolling can be detected via the key names WheelLeft and WheelRight. Some mice have a single wheel which can be scrolled up and down or tilted left and right. Generally in those cases, WheelLeft or WheelRight signals are sent repeatedly while the wheel is held to one side, to simulate continuous scrolling. This typically causes the hotkeys to execute repeatedly.

The built-in variable A_EventInfo contains the amount by which the wheel was turned, which is typically 120. However, A_EventInfo can be greater or less than 120 under the following circumstances:

If the mouse hardware reports distances of less than one notch, A_EventInfo may be less than 120;

If the wheel is being turned quickly (depending on the type of mouse), A_EventInfo may be greater than 120. A hotkey like the following can help analyze your mouse: ~WheelDown::ToolTip A_EventInfo

Some of the most useful hotkeys for the mouse wheel involve alternate modes of scrolling a window's text. For example, the following pair of hotkeys scrolls horizontally instead of vertically when you turn the wheel while holding down the left Ctrl:

```
~LControl & WheelUp::  ; Scroll left.
{
   Loop 2  ; <-- Increase this value to scroll faster.
      SendMessage 0x0114, 0, 0, ControlGetFocus("A")  ; 0x0114 is WM_HSCROLL and the 0 after it is SB_LINELEFT.
}

~LControl & WheelDown::  ; Scroll right.
{
   Loop 2  ; <-- Increase this value to scroll faster.
      SendMessage 0x0114, 1, 0, ControlGetFocus("A")  ; 0x0114 is WM_HSCROLL and the 1 after it is SB_LINERIGHT.
}
```

Finally, since mouse wheel hotkeys generate only down-events (never up-events), they cannot be used as key-up hotkeys.

Hotkey Tips and Remarks

Each numpad key can be made to launch two different hotkey subroutines depending on the state of NumLock. Alternatively, a numpad key can be made to launch the same subroutine regardless of the state. For example:

```
NumpadEnd::

Numpad1::

{

   MsgBox "This hotkey is launched regardless of whether NumLock is on."

}
```

If the tilde (~) operator is used with a prefix key even once, it changes the behavior of that prefix key for all combinations. For example, in both of the below hotkeys, the active window will receive all right-clicks even though only one of the definitions contains a tilde:

```
~RButton & LButton::MsgBox "You pressed the left mouse button while holding down the right."

RButton & WheelUp::MsgBox "You turned the mouse wheel up while holding down the right button."
```

The Suspend function can temporarily disable all hotkeys except for ones you make exempt. For greater selectivity, use #HotIf.

By means of the Hotkey function, hotkeys can be created dynamically while the script is running. The Hotkey function can also modify, disable, or enable the script's existing hotkeys individually.

Joystick hotkeys do not currently support modifier prefixes such as ^ (Ctrl) and # (Win). However, you can use GetKeyState to mimic this effect as shown in the following example:

```
Joy2::

{

   if not GetKeyState("Control")  ; Neither the left nor right Control key is down.

      return  ; i.e. Do nothing.

   MsgBox "You pressed the first joystick's second button while holding down the Control key."

}
```

There may be times when a hotkey should wait for its own modifier keys to be released before continuing. Consider the following example:

```
^!s::Send "{Delete}"
```

Pressing Ctrl+Alt+S would cause the system to behave as though you pressed Ctrl+Alt+Del (due to the system's aggressive detection of this hotkey). To work around this, use KeyWait to wait for the keys to be released; for example:

```
^!s::
{
   KeyWait "Control"
   KeyWait "Alt"
   Send "{Delete}"
}
```

If a hotkey like #z:: produces an error like "Invalid Hotkey", your system's keyboard layout/language might not have the specified character ("Z" in this case). Try using a different character that you know exists in your keyboard layout.

A hotkey's function can be called explicitly by the script only if the function has been named. See Named Function Hotkeys.

One common use for hotkeys is to start and stop a repeating action, such as a series of keystrokes or mouse clicks. For an example of this, see this FAQ topic.

Finally, each script is quasi multi-threaded, which allows a new hotkey to be launched even when a previous hotkey subroutine is still running. For example, new hotkeys can be launched even while a message box is being displayed by the current hotkey.

Alt-Tab Hotkeys

Alt-Tab hotkeys simplify the mapping of new key combinations to the system's Alt-Tab hotkeys, which are used to invoke a menu for switching tasks (activating windows).

Each Alt-Tab hotkey must be either a single key or a combination of two keys, which is typically achieved via the ampersand symbol (&). In the following example, you would hold down the right Alt and press J or K to navigate the alt-tab menu:

```
RAlt & j::AltTab

RAlt & k::ShiftAltTab
```

AltTab and ShiftAltTab are two of the special commands that are only recognized when used on the same line as a hotkey. Here is the complete list:

AltTab: If the alt-tab menu is visible, move forward in it. Otherwise, display the menu (only if the hotkey is a combination of two keys; otherwise, it does nothing).

ShiftAltTab: Same as above except move backward in the menu.

AltTabMenu: Show or hide the alt-tab menu.

AltTabAndMenu: If the alt-tab menu is visible, move forward in it. Otherwise, display the menu.

AltTabMenuDismiss: Close the Alt-tab menu.

To illustrate the above, the mouse wheel can be made into an entire substitute for Alt-tab. With the following hotkeys in effect, clicking the middle button displays the menu and turning the wheel navigates through it:

```
MButton::AltTabMenu

WheelDown::AltTab

WheelUp::ShiftAltTab
```

To cancel the Alt-Tab menu without activating the selected window, press or send Esc. In the following example, you would hold the left Ctrl and press CapsLock to display the menu and advance forward in it. Then you would release the left Ctrl to activate the selected window, or press the mouse wheel to cancel. Define the AltTabWindow window group as shown below before running this example.

```
LCtrl & CapsLock::AltTab

#HotIf WinExist("ahk_group AltTabWindow")  ; Indicates that the alt-tab menu is present on the screen.

*MButton::Send "{Blind}{Escape}"  ; The * prefix allows it to fire whether or not Alt is held down.

#HotIf
```

If the script sent {Alt Down} (such as to invoke the Alt-Tab menu), it might also be necessary to send {Alt Up} as shown in the example further below.

General Remarks

Currently, all special Alt-tab actions must be assigned directly to a hotkey as in the examples above (i.e. they cannot be used as though they were functions). They are not affected by #HotIf.

An alt-tab action may take effect on key-down and/or key-up regardless of whether the up keyword is used, and cannot be combined with another action on the same key. For example, using both F1::AltTabMenu and F1 up::OtherAction() is unsupported.

Custom alt-tab actions can also be created via hotkeys. As the identity of the alt-tab menu differs between OS versions, it may be helpful to use a window group as shown below. For the examples above and below which use ahk_group AltTabWindow, this window group is expected to be defined during script startup. Alternatively, ahk_group AltTabWindow can be replaced with the appropriate ahk_class for your system.

```
GroupAdd "AltTabWindow", "ahk_class MultitaskingViewFrame"  ; Windows 10

GroupAdd "AltTabWindow", "ahk_class TaskSwitcherWnd"  ; Windows Vista, 7, 8.1

GroupAdd "AltTabWindow", "ahk_class #32771"  ; Older, or with classic alt-tab enabled
```

In the following example, you would press F1 to display the menu and advance forward in it. Then you would press F2 to activate the selected window, or press Esc to cancel:

```
*F1::Send "{Alt down}{tab}" ; Asterisk is required in this case.

!F2::Send "{Alt up}"  ; Release the Alt key, which activates the selected window.

#HotIf WinExist("ahk_group AltTabWindow")

~*Esc::Send "{Alt up}"  ; When the menu is cancelled, release the Alt key automatically.

;*Esc::Send "{Esc}{Alt up}"  ; Without tilde (~), Escape would need to be sent.

#HotIf
```

Named Function Hotkeys

If the function of a hotkey is ever needed to be called without triggering the hotkey itself, one or more hotkeys can be assigned a named function by simply defining it immediately after the hotkey's double-colon as in this example:

```
; Ctrl+Shift+O to open containing folder in Explorer.

; Ctrl+Shift+E to open folder with current file selected.

; Supports SciTE and Notepad++.

^+o::

^+e::

   editor_open_folder(hk)

   {

      path := WinGetTitle("A")

      if RegExMatch(path, "\*?\K(.*)\\[^\\]+(?= [-*] )", &path)

         if (FileExist(path[0]) && hk = "^+e")

            Run Format('explorer.exe /select,"{1}"', path[0])

         else

            Run Format('explorer.exe "{1}"', path[1])
```

```
}
```

If the function editor_open_folder is ever called explicitly by the script, the first parameter (hk) must be passed a value.

Hotstrings can also be defined this way. Multiple hotkeys or hotstrings can be stacked together to call the same function.

There must only be whitespace or comments between the hotkey and the function name.

Naming the function also encourages self-documenting hotkeys, like in the code above where the function name describes the hotkey.

The Hotkey function can also be used to assign a function or function object to a hotkey.

Introduction and Simple Examples

Although hotstrings are mainly used to expand abbreviations as you type them (auto-replace), they can also be used to launch any scripted action. In this respect, they are similar to hotkeys except that they are typically composed of more than one character (that is, a string).

To define a hotstring, enclose the triggering abbreviation between pairs of colons as in this example:

```
::btw::by the way
```

In the above example, the abbreviation btw will be automatically replaced with "by the way" whenever you type it (however, by default you must type an ending character after typing btw, such as Space, ., or Enter).

The "by the way" example above is known as an auto-replace hotstring because the typed text is automatically erased and replaced by the string specified after the second pair of colons. By contrast, a hotstring may also be defined to perform any custom action as in the following examples. Note that the functions must appear beneath the hotstring:

```
::btw::
{
   MsgBox 'You typed "btw".'
}

:*:]d::  ; This hotstring replaces "]d" with the current date and time via the functions below.
{
   SendInput FormatTime(, "M/d/yyyy h:mm tt")  ; It will look like 9/1/2005 3:53 PM
}
```

In the above, the braces serve to define a function body for each hotstring. The opening brace may also be specified on the same line as the double-colon to support the OTB (One True Brace) style.

Even though the two examples above are not auto-replace hotstrings, the abbreviation you type is erased by default. This is done via automatic backspacing, which can be disabled via the b0 option.

When a hotstring is triggered, the name of the hotstring is passed as its first parameter named ThisHotkey (which excludes the trailing colons). For example:

```
:X:btw::MsgBox ThisHotkey  ; Reports :X:btw
```

With few exceptions, this is similar to the built-in variable A_ThisHotkey. The parameter name can be changed by using a named function.

Ending Characters

Unless the asterisk option is in effect, you must type an ending character after a hotstring's abbreviation to trigger it. Ending characters initially consist of the following: -()[]{}':;"/\,.?!`n `t (note that `n is Enter, `t is Tab, and there is a plain space between `n and `t). This set of characters can be changed by editing the following example, which sets the new ending characters for all hotstrings, not just the ones beneath it:

```
#Hotstring EndChars -()[]{}:;'"/\,.?!`n`s`t
```

The ending characters can be changed while the script is running by calling the Hotstring function as demonstrated below:

```
Hotstring("EndChars", "-()[]{}:;")
```

Options

A hotstring's default behavior can be changed in two possible ways:

The #Hotstring directive, which affects all hotstrings physically beneath that point in the script. The following example puts the C and R options into effect: #Hotstring c r.

Putting options inside a hotstring's first pair of colons. The following example puts the C and * options (case sensitive and "ending character not required") into effect for a single hotstring: :c*:j@::john@somedomain.com.

The list below describes each option. When specifying more than one option using the methods above, spaces optionally may be included between them.

* (asterisk): An ending character (e.g. Space, ., or Enter) is not required to trigger the hotstring. For example:

:*:j@::jsmith@somedomain.com

The example above would send its replacement the moment you type the @ character. When using the #Hotstring directive, use *0 to turn this option back off.

? (question mark): The hotstring will be triggered even when it is inside another word; that is, when the character typed immediately before it is alphanumeric. For example, if :?:al::airline is a hotstring, typing "practical " would produce "practicairline ". Use ?0 to turn this option back off.

B0 (B followed by a zero): Automatic backspacing is not done to erase the abbreviation you type. Use a plain B to turn backspacing back on after it was previously turned off. A script may also do its own backspacing via {bs 5}, which sends Backspace five times. Similarly, it may send ← five times via {left 5}. For example, the following hotstring produces "" and moves the caret 5 places to the left (so that it's between the tags):

:*b0:::{left 5}

C: Case sensitive: When you type an abbreviation, it must exactly match the case defined in the script. Use C0 to turn case sensitivity back off.

C1: Do not conform to typed case. Use this option to make auto-replace hotstrings case insensitive and prevent them from conforming to the case of the characters you actually type. Case-conforming hotstrings (which are the default) produce their replacement text in all caps if you type the abbreviation in all caps. If you type the first letter in caps, the first letter of the replacement will also be capitalized (if it is a letter). If you type the case in any other way, the replacement is sent exactly as defined. When using the #Hotstring directive, C0 can be used to turn this option back off, which makes hotstrings conform again.

Kn: Key-delay: This rarely-used option sets the delay between keystrokes produced by auto-backspacing or auto-replacement. Specify the new delay for n; for example, specify k10 to have a 10ms delay and k-1 to have no delay. The exact behavior of this option depends on which sending mode is in effect:

SI (SendInput): Key-delay is ignored because a delay is not possible in this mode. The exception to this is when SendInput is unavailable, in which case hotstrings revert to SendPlay mode below (which does obey key-delay).

SP (SendPlay): A delay of length zero is the default, which for SendPlay is the same as -1 (no delay). In this mode, the delay is actually a PressDuration rather than a delay between keystrokes.

SE (SendEvent): A delay of length zero is the default. Zero is recommended for most purposes since it is fast but still cooperates well with other processes (due to internally doing a Sleep 0). Specify k-1 to have no delay at all, which is useful to make auto-replacements faster if your CPU is frequently under heavy load. When set to -1, a script's process-priority becomes an important factor in how fast it can send keystrokes. To raise a script's priority, use ProcessSetPriority "High".

O: Omit the ending character of auto-replace hotstrings when the replacement is produced. This is useful when you want a hotstring to be kept unambiguous by still requiring an ending character, but don't actually want the ending character to be shown on the screen. For example, if :o:ar::aristocrat is a hotstring, typing "ar" followed by the spacebar will produce "aristocrat" with no trailing space, which allows you to make the word plural or possessive without having to press Backspace. Use O0 (the letter O followed by a zero) to turn this option back off.

Pn: The priority of the hotstring (e.g. P1). This rarely-used option has no effect on auto-replace hotstrings.

R: Send the replacement text raw; that is, without translating {Enter} to Enter, ^c to Ctrl+C, etc. Use R0 to turn this option back off, or override it with T.

Note: Text mode may be more reliable. The R and T options are mutually exclusive.

S or S0: Specify the letter S to make the hotstring exempt from Suspend. Specify S0 (S with the number 0) to remove the exemption, allowing the hotstring to be suspended. When applied as a default option, either S or #SuspendExempt will make the hotstring exempt; that is, to override the directive, S0 must be used explicitly in the hotstring.

SI or SP or SE: Sets the method by which auto-replace hotstrings send their keystrokes. These options are mutually exclusive: only one can be in effect at a time. The following describes each option:

SI stands for SendInput, which typically has superior speed and reliability than the other modes. Another benefit is that like SendPlay below, SendInput postpones anything you type during a hotstring's auto-replacement text. This prevents your keystrokes from being interspersed with those of the replacement. When SendInput is unavailable, hotstrings automatically use SendPlay instead.

SP stands for SendPlay, which may allow hotstrings to work in a broader variety of games.

SE stands for SendEvent.

If none of the above options are used, the default mode is SendInput. However, unlike the SI option, SendEvent is used instead of SendPlay when SendInput is unavailable.

T: Send the replacement text using Text mode. That is, send each character by character code, without translating {Enter} to Enter, ^c to Ctrl+C, etc. and without translating each character to a keystroke. This option is put into effect automatically for hotstrings that have a continuation section. Use T0 or R0 to turn this option back off, or override it with R.

X: Execute. Instead of replacement text, the hotstring accepts a function call or expression to execute. For example, :X:~mb::MsgBox would cause a message box to be displayed when the user types "~mb" instead of auto-replacing it with the word "MsgBox". This is most useful when defining a large number of hotstrings which call functions, as it would otherwise require three lines per hotstring.

This option should not be used with the Hotstring function. To make a hotstring call a function when triggered, pass the function by reference.

Z: This rarely-used option resets the hotstring recognizer after each triggering of the hotstring. In other words, the script will begin waiting for an entirely new hotstring, eliminating from consideration anything you previously typed. This can prevent unwanted triggerings of hotstrings. To illustrate, consider the following hotstring:

```
:b0*?:11::
{
   SendInput "xx"
}
```

Since the above lacks the Z option, typing 111 (three consecutive 1's) would trigger the hotstring twice because the middle 1 is the last character of the first triggering but also the first character of the second triggering. By adding the letter Z in front of b0, you would have to type four 1's instead of three to trigger the hotstring twice. Use Z0 to turn this option back off.

Long Replacements

Hotstrings that produce a large amount of replacement text can be made more readable and maintainable by using a continuation section. For example:

```
::text1::
(
Any text between the top and bottom parentheses is treated literally.

By default, the hard carriage return (Enter) between the previous line and this one is also preserved.

   By default, the indentation (tab) to the left of this line is preserved.
)
```

See continuation section for how to change these default behaviors. The presence of a continuation section also causes the hotstring to default to Text mode. The only way to override this special default is to specify an opposing option in each hotstring that has a continuation section (e.g. :t0:text1:: or :r:text2::).

Context-sensitive Hotstrings

The #HotIf directive can be used to make selected hotstrings context sensitive. Such hotstrings send a different replacement, perform a different action, or do nothing at all depending on any condition, such as the type of window that is active. For example:

#HotIf WinActive("ahk_class Notepad")

::btw::This replacement text will appear only in Notepad.

#HotIf

::btw::This replacement text appears in windows other than Notepad.

AutoCorrect

The following script uses hotstrings to correct about 4700 common English misspellings on-the-fly. It also includes a Win+H hotkey to make it easy to add more misspellings:

Download: AutoCorrect.ahk (127 KB)

Author: Jim Biancolo and Wikipedia's Lists of Common Misspellings

Remarks

Expressions are not currently supported within the replacement text. To work around this, don't make such hotstrings auto-replace. Instead, use the SendInput function beneath the abbreviation, followed by a line containing only the word Return.

To send an extra space or tab after a replacement, include the space or tab at the end of the replacement but make the last character an accent/backtick (`). For example:

:*:btw::By the way `

For an auto-replace hotstring which doesn't use the Text or Raw mode, sending a { alone, or one preceded only by white-space, requires it being enclosed in a pair of brackets, for example :*:brace::{{} and :*:space_brace:: {{}. Otherwise it is interpreted as the opening brace for the hotstring's function to support the OTB (One True Brace) style.

By default, any click of the left or right mouse button will reset the hotstring recognizer. In other words, the script will begin waiting for an entirely new hotstring, eliminating from consideration anything you previously typed (if this is undesirable, specify the line #Hotstring NoMouse

anywhere in the script). This "reset upon mouse click" behavior is the default because each click typically moves the text insertion point (caret) or sets keyboard focus to a new control/field. In such cases, it is usually desirable to: 1) fire a hotstring even if it lacks the question mark option; 2) prevent a firing when something you type after clicking the mouse accidentally forms a valid abbreviation with what you typed before.

The hotstring recognizer checks the active window each time a character is typed, and resets if a different window is active than before. If the active window changes but reverts before any characters are typed, the change is not detected (but the hotstring recognizer may be reset for some other reason). The hotstring recognizer can also be reset by calling Hotstring "Reset".

The built-in variable A_EndChar contains the ending character that you typed to trigger the most recent non-auto-replace hotstring. If no ending character was required (due to the * option), it will be blank. A_EndChar is useful when making hotstrings that use the Send function or whose behavior should vary depending on which ending character you typed. To send the ending character itself, use SendText A_EndChar (SendText is used because characters such as !{} would not be sent correctly by the normal Send function).

Although single-colons within hotstring definitions do not need to be escaped unless they precede the double-colon delimiter, backticks and those semicolons having a space or tab to their left must always be escaped. See Escape Sequences for a complete list.

Although the Send function's special characters such as {Enter} are supported in auto-replacement text (unless the raw option is used), the hotstring abbreviations themselves do not use this. Instead, specify `n for Enter and `t (or a literal tab) for Tab (see Escape Sequences for a complete list). For example, the hotstring :*:ab`t:: would be triggered when you type "ab" followed by a tab.

Spaces and tabs are treated literally within hotstring definitions. For example, the following would produce two different results: ::btw::by the way and ::btw:: by the way.

Each hotstring abbreviation can be no more than 40 characters long. The program will warn you if this length is exceeded. By contrast, the length of hotstring's replacement text is limited to about 5000 characters when the sending mode is at its default of SendInput. That limit can be removed by switching to one of the other sending modes, or by using SendPlay or SendEvent in the body of the hotstring.

The order in which hotstrings are defined determines their precedence with respect to each other. In other words, if more than one hotstring matches something you type, only the one listed first in the script will take effect. Related topic: context-sensitive hotstrings.

Any backspacing you do is taken into account for the purpose of detecting hotstrings. However, the use of ↑, →, ↓, ←, PgUp, PgDn, Home, and End to navigate within an editor will cause the hotstring recognition process to reset. In other words, it will begin waiting for an entirely new hotstring.

A hotstring may be typed even when the active window is ignoring your keystrokes. In other words, the hotstring will still fire even though the triggering abbreviation is never visible. In addition, you may still press Backspace to undo the most recently typed keystroke (even though you can't see the effect).

A hotstring's function can be called explicitly by the script only if the function has been named. See Named Function Hotstrings.

Hotstrings are not monitored and will not be triggered while input is blocked by an invisible Input hook.

By default, hotstrings are never triggered by keystrokes produced by any AutoHotkey script. This avoids the possibility of an infinite loop where hotstrings trigger each other over and over. This behaviour can be controlled with #InputLevel and SendLevel. However, auto-replace hotstrings always use send level 0 and therefore never trigger hook hotkeys or hotstrings.

Hotstrings can be created dynamically by means of the Hotstring function, which can also modify, disable, or enable the script's existing hotstrings individually.

The InputHook function is more flexible than hotstrings for certain purposes. For example, it allows your keystrokes to be invisible in the active window (such as a game). It also supports non-character ending keys such as Esc.

The keyboard hook is automatically used by any script that contains hotstrings.

Hotstrings behave identically to hotkeys in the following ways:

They are affected by the Suspend function.

They obey #MaxThreads and #MaxThreadsPerHotkey (but not #MaxThreadsBuffer).

Scripts containing hotstrings are automatically persistent.

Non-auto-replace hotstrings will create a new thread when launched. In addition, they will update the built-in hotkey variables such as A_ThisHotkey.

Known limitation: On some systems in Java applications, hotstrings might interfere with the user's ability to type diacritical letters (via dead keys). To work around this, Suspend can be turned on temporarily (which disables all hotstrings).

Named Function Hotstrings

If the function of a hotstring is ever needed to be called without triggering the hotstring itself, one or more hotstrings can be assigned a named function by simply defining it immediately after the hotstring's double-colon, as in this example:

```
; This example also demonstrates one way to implement case conformity in a script.
:C:BTW::  ; Typed in all-caps.
:C:Btw::  ; Typed with only the first letter upper-case.
: :btw::  ; Typed in any other combination.
   case_conform_btw(hs) ; hs will hold the name of the hotstring which triggered the function.
   {
      if (hs == ":C:BTW")
         Send "BY THE WAY"
      else if (hs == ":C:Btw")
         Send "By the way"
```

```
    else
        Send "by the way"
}
```

If the function case_conform_btw is ever called explicitly by the script, the first parameter (hs) must be passed a value.

Hotkeys can also be defined this way. Multiple hotkeys or hotstrings can be stacked together to call the same function.

There must only be whitespace or comments between the hotstring and the function name.

Naming the function also encourages self-documenting hotstrings, like in the code above where the function name describes the hotstring.

The Hotstring function can also be used to assign a function or function object to a hotstring.

Hotstring Helper

Take a look at the first example in the example section of the Hotstring function's page, which might be useful if you are a heavy user of hotstrings. By pressing Win+H (or another hotkey of your choice), the currently selected text can be turned into a hotstring. For example, if you have "by the way" selected in a word processor, pressing Win+H will prompt you for its abbreviation (e.g. btw), add the new hotstring to the script and activate it.

Remapping Keys (Keyboard, Mouse and Joystick)

Table of Contents

Remarks

Moving the Mouse Cursor via the Keyboard

Remapping via the Registry's "Scancode Map"

Related Topics

Introduction

Limitation: AutoHotkey's remapping feature described below is generally not as pure and effective as remapping directly via the Windows registry. For the advantages and disadvantages of each approach, see registry remapping.

Remapping the Keyboard and Mouse

The syntax for the built-in remapping feature is OriginKey::DestinationKey. For example, a script consisting only of the following line would make A behave like B:

```
a::b
```

The above example does not alter B itself. B would continue to send the "b" keystroke unless you remap it to something else as shown in the following example:

```
a::b

b::a
```

The examples above use lowercase, which is recommended for most purposes because it also remaps the corresponding uppercase letters (that is, it will send uppercase when CapsLock is "on" or Shift is held down). By contrast, specifying an uppercase letter on the right side forces uppercase. For example, the following line would produce an uppercase B when you type either "a" or "A" (as long as CapsLock is off):

```
a::B
```

Conversely, any modifiers included on the left side but not the right side are automatically released when the key is sent. For example, the following two lines would produce a lowercase "b" when you press either Shift+A or Ctrl+A:

A::b

^a::b

Mouse Remapping

To remap the mouse instead of the keyboard, use the same approach. For example:

Example	Description
MButton::Shift	Makes the middle button behave like Shift.
XButton1::LButton	Makes the fourth mouse button behave like the left mouse button.
RAlt::RButton	Makes the right Alt behave like the right mouse button.

Other Useful Remappings

Example	Description
CapsLock::Ctrl	Makes CapsLock become Ctrl. To retain the ability to turn CapsLock on and off, add the remapping +CapsLock::CapsLock first. This toggles CapsLock on and off when you hold down Shift and press CapsLock. Because both remappings allow additional modifier keys to be held down, the more specific +CapsLock::CapsLock remapping must be placed first for it to work.
XButton2::^LButton	Makes the fifth mouse button (XButton2) produce a control-click.
RAlt::AppsKey	Makes the right Alt become Menu (which is the key that opens the context menu).
RCtrl::RWin	Makes the right Ctrl become the right Win.
Ctrl::Alt	Makes both Ctrl behave like Alt. However, see alt-tab issues.
^x::^c	Makes Ctrl+X produce Ctrl+C. It also makes Ctrl+Alt+X produce Ctrl+Alt+C, etc.
RWin::Return	Disables the right Win by having it simply return.

You can try out any of these examples by copying them into a new text file such as "Remap.ahk", then launching the file.

See the Key List for a complete list of key and mouse button names.

Remarks

The #HotIf directive can be used to make selected remappings active only in the windows you specify (or while any given condition is met). For example:

```
#HotIf WinActive("ahk_class Notepad")

a::b  ; Makes the 'a' key send a 'b' key, but only in Notepad.

#HotIf  ; This puts subsequent remappings and hotkeys in effect for all windows.
```

Remapping a key or button is "complete" in the following respects:

Holding down a modifier such as Ctrl or Shift while typing the origin key will put that modifier into effect for the destination key. For example, b::a would produce Ctrl+A if you press Ctrl+B.

CapsLock generally affects remapped keys in the same way as normal keys.

The destination key or button is held down for as long as you continue to hold down the origin key. However, some games do not support remapping; in such cases, the keyboard and mouse will behave as though not remapped.

Remapped keys will auto-repeat while being held down (except keys remapped to become mouse buttons).

Although a remapped key can trigger normal hotkeys, by default it cannot trigger mouse hotkeys or hook hotkeys (use ListHotkeys to discover which hotkeys are "hook"). For example, if the remapping a::b is in effect, pressing Ctrl+Alt+A would trigger the ^!b hotkey only if ^!b is not a hook hotkey. If ^!b is a hook hotkey, you can define ^!a as a hotkey if you want Ctrl+Alt+A to perform the same action as Ctrl+Alt+B. For example:

```
a::b

^!a::

^!b::ToolTip "You pressed " ThisHotkey
```

Alternatively, #InputLevel can be used to override the default behaviour. For example:

```
#InputLevel 1

a::b
```

#InputLevel 0

^!b::ToolTip "You pressed " ThisHotkey

If SendMode is used during script startup, it affects all remappings. However, since remapping uses Send "{Blind}" and since the SendPlay mode does not fully support {Blind}, some remappings might not function properly in SendPlay mode (especially Ctrl, Shift, Alt, and Win). To work around this, avoid using SendMode "Play" during script startup when you have remappings; then use the function SendPlay vs. Send in other places throughout the script. Alternatively, you could translate your remappings into hotkeys (as described below) that explicitly call SendEvent vs. Send.

If DestinationKey is meant to be {, it has to be escaped, for example, x::`{. Otherwise it is interpreted as the opening brace for the hotkey's function.

When a script is launched, each remapping is translated into a pair of hotkeys. For example, a script containing a::b actually contains the following two hotkeys instead:

*a::

{

SetKeyDelay -1 ; If the destination key is a mouse button, SetMouseDelay is used instead.

Send "{Blind}{b DownR}" ; DownR is like Down except that other Send functions in the script won't assume "b" should stay down during their Send.

}

*a up::

{

SetKeyDelay -1 ; See note below for why press-duration is not specified with either of these SetKeyDelays.

Send "{Blind}{b Up}"

}

However, the above hotkeys vary under the following circumstances:

When the source key is the left Ctrl and the destination key is Alt, the line Send "{Blind}{LAlt DownR}" is replaced by Send "{Blind}{LCtrl up}{LAlt DownR}". The same is true if the source is the right Ctrl, except that {RCtrl up} is used.

When a keyboard key is being remapped to become a mouse button (e.g. RCtrl::RButton), the hotkeys above use SetMouseDelay in place of SetKeyDelay. In addition, the first hotkey above is replaced by the following, which prevents the keyboard's auto-repeat feature from generating repeated mouse clicks:

```
*RCtrl::
{
   SetMouseDelay -1
   if not GetKeyState("RButton")  ; i.e. the right mouse button isn't down yet.
      Send "{Blind}{RButton DownR}"
}
```

When the source is a custom combination, the wildcard modifier (*) is omitted to allow the hotkeys to work.

Note that SetKeyDelay's second parameter (press duration) is omitted in the hotkeys above. This is because press-duration does not apply to down-only or up-only events such as {b down} and {b up}. However, it does apply to changes in the state of the modifier keys (Shift, Ctrl, Alt, and Win), which affects remappings such as a::B or a::^b. Consequently, any press-duration a script puts into effect during script startup will apply to all such remappings.

Since remappings are translated into hotkeys as described above, the Suspend function affects them. Similarly, the Hotkey function can disable or modify a remapping. For example, the following two functions would disable the remapping a::b.

```
Hotkey "*a", "Off"
Hotkey "*a up", "Off"
```

Alt-tab issues: If you remap a key or mouse button to become Alt, that key will probably not be able to alt-tab properly. A possible work-around is to add the hotkey *Tab::Send "{Blind}{Tab}" -- but be aware that it will likely interfere with using the real Alt to alt-tab. Therefore, it should be used only when you alt-tab solely by means of remapped keys and/or alt-tab hotkeys.

In addition to the keys and mouse buttons on the Key List page, the source key may also be a virtual key (VKnn) or scan code (SCnnn) as described on the special keys page. The same is true for the destination key except that it may optionally specify a scan code after the virtual key. For example, sc01e::vk42sc030 is equivalent to a::b on most keyboard layouts.

To disable a key rather than remapping it, make it a hotkey that simply returns. For example, F1::return would disable F1.

The following keys are not supported by the built-in remapping method:

The mouse wheel (WheelUp/Down/Left/Right).

"Pause" as a destination key name (since it matches the name of a built-in function). Instead use vk13 or the corresponding scan code.

Curly braces {} as destination keys. Instead use the VK/SC method; e.g. x::+sc01A and y::+sc01B.

Moving the Mouse Cursor via the Keyboard

The keyboard can be used to move the mouse cursor as demonstrated by the fully-featured Keyboard-To-Mouse script. Since that script offers smooth cursor movement, acceleration, and other features, it is the recommended approach if you plan to do a lot of mousing with the keyboard. By contrast, the following example is a simpler demonstration:

```
*#up::MouseMove 0, -10, 0, "R"  ; Win+UpArrow hotkey => Move cursor upward

*#Down::MouseMove 0, 10, 0, "R"  ; Win+DownArrow => Move cursor downward

*#Left::MouseMove -10, 0, 0, "R"  ; Win+LeftArrow => Move cursor to the left

*#Right::MouseMove 10, 0, 0, "R"  ; Win+RightArrow => Move cursor to the right

*<#RCtrl::  ; LeftWin + RightControl => Left-click (hold down Control/Shift to Control-Click or Shift-Click).

{

   SendEvent "{Blind}{LButton down}"

   KeyWait "RCtrl"  ; Prevents keyboard auto-repeat from repeating the mouse click.
```

```
   SendEvent "{Blind}{LButton up}"
}

*<#AppsKey::  ; LeftWin + AppsKey => Right-click
{
   SendEvent "{Blind}{RButton down}"
   KeyWait "AppsKey"  ; Prevents keyboard auto-repeat from repeating the mouse click.
   SendEvent "{Blind}{RButton up}"
}
```

Remapping via the Registry's "Scancode Map"

Advantages:

Registry remapping is generally more pure and effective than AutoHotkey's remapping. For example, it works in a broader variety of games, it has no known alt-tab issues, and it is capable of firing AutoHotkey's hook hotkeys (whereas AutoHotkey's remapping requires a workaround).

If you choose to make the registry entries manually (explained below), absolutely no external software is needed to remap your keyboard. Even if you use KeyTweak to make the registry entries for you, KeyTweak does not need to stay running all the time (unlike AutoHotkey).

Disadvantages:

Registry remapping is relatively permanent: a reboot is required to undo the changes or put new ones into effect.

Its effect is global: it cannot create remappings specific to a particular user, application, or locale.

It cannot send keystrokes that are modified by Shift, Ctrl, Alt, or AltGr. For example, it cannot remap a lowercase character to an uppercase one.

It supports only the keyboard (AutoHotkey has mouse remapping and some limited joystick remapping).

How to Apply Changes to the Registry: There are at least two methods to remap keys via the registry:

Use a program like KeyTweak (freeware) to visually remap your keys. It will change the registry for you.

Remap keys manually by creating a .reg file (plain text) and loading it into the registry. This is demonstrated in the archived forums.

Related Topics

List of keys and mouse buttons

GetKeyState

Remapping a joystick

List of Keys (Keyboard, Mouse and Joystick)

Table of Contents

CapsLock and IME

Mouse

General Buttons

Name Description

LButton Primary mouse button. Which physical button this corresponds to depends on system settings; by default it is the left button.

RButton Secondary mouse button. Which physical button this corresponds to depends on system settings; by default it is the right button.

MButton Middle or wheel mouse button

Advanced Buttons

Name Description

XButton1 4th mouse button. Typically performs the same function as Browser_Back.

XButton2 5th mouse button. Typically performs the same function as Browser_Forward.

Wheel

Name Description

WheelDown Turn the wheel downward (toward you).

WheelUp Turn the wheel upward (away from you).

WheelLeft

WheelRight

Scroll to the left or right.

These can be used as hotkeys with some (but not all) mice which have a second wheel or support tilting the wheel to either side. In some cases, software bundled with the mouse must instead be used to control this feature. Regardless of the particular mouse, Send and Click can be used to scroll horizontally in programs which support it.

Keyboard

Note: The names of the letter and number keys are the same as that single letter or digit. For example: b is B and 5 is 5.

Although any single character can be used as a key name, its meaning (scan code or virtual keycode) depends on the current keyboard layout. Additionally, some special characters may need to be escaped or enclosed in braces, depending on the context. The letters a-z or A-Z can be used to refer to the corresponding virtual keycodes (usually vk41-vk5A) even if they are not included in the current keyboard layout.

General Keys

Name Description

CapsLock CapsLock (caps lock key)

Note: Windows IME may interfere with the detection and functionality of CapsLock; see CapsLock and IME for details.

Space Space (space bar)

Tab Tab (tabulator key)

Enter Enter

Escape (or Esc) Esc

Backspace (or BS) Backspace

Cursor Control Keys

Name Description

ScrollLock ScrollLock (scroll lock key). While Ctrl is held down, ScrollLock produces the key code of CtrlBreak, but can be differentiated from Pause by scan code.

Delete (or Del) Del

Insert (or Ins) Ins

Home Home

End End

PgUp PgUp (page up key)

PgDn PgDn (page down key)

Up ↑ (up arrow key)

Down ↓ (down arrow key)

Left ← (left arrow key)

Right → (right arrow key)

Numpad Keys

Due to system behavior, the following keys separated by a slash are identified differently depending on whether NumLock is ON or OFF. If NumLock is OFF but Shift is pressed, the system temporarily releases Shift and acts as though NumLock is ON.

Name	Description
Numpad0 / NumpadIns	0 / Ins
Numpad1 / NumpadEnd	1 / End
Numpad2 / NumpadDown	2 / ↓
Numpad3 / NumpadPgDn	3 / PgDn
Numpad4 / NumpadLeft	4 / ←
Numpad5 / NumpadClear	5 / typically does nothing
Numpad6 / NumpadRight	6 / →
Numpad7 / NumpadHome	7 / Home
Numpad8 / NumpadUp	8 / ↑
Numpad9 / NumpadPgUp	9 / PgUp
NumpadDot / NumpadDel	. / Del

NumLock NumLock (number lock key). While Ctrl is held down, NumLock produces the key code of Pause, so use ^Pause in hotkeys instead of ^NumLock.

NumpadDiv / (division)

NumpadMult * (multiplication)

NumpadAdd + (addition)

NumpadSub - (subtraction)

NumpadEnter Enter

Function Keys

Name Description

F1 - F24 The 12 or more function keys at the top of most keyboards.

Modifier Keys

Name Description

LWin Left Win. Corresponds to the <# hotkey prefix.

RWin

Right Win. Corresponds to the ># hotkey prefix.

Note: Unlike Ctrl/Alt/Shift, there is no generic/neutral "Win" key because the OS does not support it. However, hotkeys with the # modifier can be triggered by either Win.

Control (or Ctrl) Ctrl. As a hotkey (Control::) it fires upon release unless it has the tilde prefix. Corresponds to the ^ hotkey prefix.

Alt Alt. As a hotkey (Alt::) it fires upon release unless it has the tilde prefix. Corresponds to the ! hotkey prefix.

Shift Shift. As a hotkey (Shift::) it fires upon release unless it has the tilde prefix. Corresponds to the + hotkey prefix.

LControl (or LCtrl) Left Ctrl. Corresponds to the <^ hotkey prefix.

RControl (or RCtrl) Right Ctrl. Corresponds to the >^ hotkey prefix.

LShift Left Shift. Corresponds to the <+ hotkey prefix.

RShift Right Shift. Corresponds to the >+ hotkey prefix.

LAlt Left Alt. Corresponds to the <! hotkey prefix.

RAlt

Right Alt. Corresponds to the >! hotkey prefix.

Note: If your keyboard layout has AltGr instead of RAlt, you can probably use it as a hotkey prefix via <^>! as described here. In addition, LControl & RAlt:: would make AltGr itself into a hotkey.

Multimedia Keys

The function assigned to each of the keys listed below can be overridden by modifying the Windows registry. This table shows the default function of each key on most versions of Windows.

Name	Description
Browser_Back	Back
Browser_Forward	Forward
Browser_Refresh	Refresh
Browser_Stop	Stop
Browser_Search	Search
Browser_Favorites	Favorites
Browser_Home	Homepage
Volume_Mute	Mute the volume
Volume_Down	Lower the volume
Volume_Up	Increase the volume
Media_Next	Next Track
Media_Prev	Previous Track
Media_Stop	Stop
Media_Play_Pause	Play/Pause
Launch_Mail	Launch default e-mail program
Launch_Media	Launch default media player
Launch_App1	Launch My Computer
Launch_App2	Launch Calculator

Other Keys

Name Description

AppsKey Menu. This is the key that invokes the right-click context menu.

PrintScreen PrtSc (print screen key)

CtrlBreak Ctrl+Pause or Ctrl+ScrollLock

Pause Pause or Ctrl+NumLock. While Ctrl is held down, Pause produces the key code of CtrlBreak and NumLock produces Pause, so use ^CtrlBreak in hotkeys instead of ^Pause.

Help Help. This probably doesn't exist on most keyboards. It's usually not the same as F1.

Sleep Sleep. Note that the sleep key on some keyboards might not work with this.

SCnnn Specify for nnn the scan code of a key. Recognizes unusual keys not mentioned above. See Special Keys for details.

VKnn

Specify for nn the hexadecimal virtual key code of a key. This rarely-used method also prevents certain types of hotkeys from requiring the keyboard hook. For example, the following hotkey does not use the keyboard hook, but as a side-effect it is triggered by pressing either Home or NumpadHome:

^VK24::MsgBox "You pressed Home or NumpadHome while holding down Control."

Known limitation: VK hotkeys that are forced to use the keyboard hook, such as *VK24 or ~VK24, will fire for only one of the keys, not both (e.g. NumpadHome but not Home). For more information about the VKnn method, see Special Keys.

Warning: Only Send, GetKeyName, GetKeyVK, GetKeySC and A_MenuMaskKey support combining VKnn and SCnnn. If combined in any other case (or if any other invalid suffix is present), the key is not recognized. For example, vk1Bsc001:: raises an error.

Joystick

Joy1 through Joy32: The buttons of the joystick. To help determine the button numbers for your joystick, use this test script. Note that hotkey prefix symbols such as ^ (control) and + (shift) are not supported (though GetKeyState can be used as a substitute). Also note that the pressing of joystick buttons always "passes through" to the active window if that window is designed to detect the pressing of joystick buttons.

Although the following Joystick control names cannot be used as hotkeys, they can be used with GetKeyState:

JoyX, JoyY, and JoyZ: The X (horizontal), Y (vertical), and Z (altitude/depth) axes of the joystick.

JoyR: The rudder or 4th axis of the joystick.

JoyU and JoyV: The 5th and 6th axes of the joystick.

JoyPOV: The point-of-view (hat) control.

JoyName: The name of the joystick or its driver.

JoyButtons: The number of buttons supported by the joystick (not always accurate).

JoyAxes: The number of axes supported by the joystick.

JoyInfo: Provides a string consisting of zero or more of the following letters to indicate the joystick's capabilities: Z (has Z axis), R (has R axis), U (has U axis), V (has V axis), P (has POV control), D (the POV control has a limited number of discrete/distinct settings), C (the POV control is continuous/fine). Example string: ZRUVPD

Multiple Joysticks: If the computer has more than one joystick and you want to use one beyond the first, include the joystick number (max 16) in front of the control name. For example, 2joy1 is the second joystick's first button.

Note: If you have trouble getting a script to recognize your joystick, one person reported needing to specify a joystick number other than 1 even though only a single joystick was present. It is unclear how this situation arises or whether it is normal, but experimenting with the joystick number in the joystick test script can help determine if this applies to your system.

See Also:

Joystick remapping: Methods of sending keystrokes and mouse clicks with a joystick.

Joystick-To-Mouse script: Using a joystick as a mouse.

Hand-held Remote Controls

Respond to signals from hand-held remote controls via the WinLIRC client script.

Special Keys

If your keyboard or mouse has a key not listed above, you might still be able to make it a hotkey by using the following steps:

Ensure that at least one script is running that is using the keyboard hook. You can tell if a script has the keyboard hook by opening its main window and selecting "View->Key history" from the menu bar.

Double-click that script's tray icon to open its main window.

Press one of the "mystery keys" on your keyboard.

Select the menu item "View->Key history"

Scroll down to the bottom of the page. Somewhere near the bottom are the key-down and key-up events for your key. NOTE: Some keys do not generate events and thus will not be visible here. If this is the case, you cannot directly make that particular key a hotkey because your keyboard driver or hardware handles it at a level too low for AutoHotkey to access. For possible solutions, see further below.

If your key is detectable, make a note of the 3-digit hexadecimal value in the second column of the list (e.g. 159).

To define this key as a hotkey, follow this example:

SC159::MsgBox ThisHotkey " was pressed." ; Replace 159 with your key's value.

Also see ThisHotkey.

Reverse direction: To remap some other key to become a "mystery key", follow this example:

; Replace 159 with the value discovered above. Replace FF (if needed) with the

; key's virtual key, which can be discovered in the first column of the Key History screen.

#c::Send "{vkFFsc159}" ; See Send {vkXXscYYY} for more details.

Alternate solutions: If your key or mouse button is not detectable by the Key History screen, one of the following might help:

Reconfigure the software that came with your mouse or keyboard (sometimes accessible in the Control Panel or Start Menu) to have the "mystery key" send some other keystroke. Such a keystroke can then be defined as a hotkey in a script. For example, if you configure a mystery key to send Ctrl+F1, you can then indirectly make that key as a hotkey by using ^F1:: in a script.

Try AHKHID. You can also try searching the forum for a keywords like RawInput*, USB HID or AHKHID.

The following is a last resort and generally should be attempted only in desperation. This is because the chance of success is low and it may cause unwanted side-effects that are difficult to undo:

Disable or remove any extra software that came with your keyboard or mouse or change its driver to a more standard one such as the one built into the OS. This assumes there is such a driver for your particular keyboard or mouse and that you can live without the features provided by its custom driver and software.

CapsLock and IME

Some configurations of Windows IME (such as Japanese input with English keyboard) use CapsLock to toggle between modes. In such cases, CapsLock is suppressed by the IME and cannot be detected by AutoHotkey. However, the Alt+CapsLock, Ctrl+CapsLock and Shift+CapsLock shortcuts can be disabled with a workaround. Specifically, send a key-up to modify the state of the IME, but prevent any other effects by signalling the keyboard hook to suppress the event. The following function can be used for this purpose:

```
; The keyboard hook must be installed.
InstallKeybdHook
SendSuppressedKeyUp(key) {
   DllCall("keybd_event"
      , "char", GetKeyVK(key)
      , "char", GetKeySC(key)
      , "uint", KEYEVENTF_KEYUP := 0x2
      , "uptr", KEY_BLOCK_THIS := 0xFFC3D450)
```

}

After copying the function into a script or saving it as SendSuppressedKeyUp.ahk in a Lib folder and adding #Include <SendSuppressedKeyUp> to the script, it can be used as follows:

```
; Disable Alt+key shortcuts for the IME.
~LAlt::SendSuppressedKeyUp "LAlt"

; Test hotkey:
!CapsLock::MsgBox A_ThisHotkey

; Remap CapsLock to LCtrl in a way compatible with IME.
*CapsLock::
{
   Send "{Blind}{LCtrl DownR}"
   SendSuppressedKeyUp "LCtrl"
}
*CapsLock up::
{
   Send "{Blind}{LCtrl Up}"
}
```


Scripts

Related topics:

Using the Program: How to use AutoHotkey, in general.

Concepts and Conventions: General explanation of various concepts utilised by AutoHotkey.

Scripting Language: Specific details about syntax (how to write scripts).

Table of Contents

Introduction

Each script is a plain text file containing lines to be executed by the program (AutoHotkey.exe). A script may also contain hotkeys and hotstrings, or even consist entirely of them. However, in the absence of hotkeys and hotstrings, a script will perform its functions sequentially from top to bottom the moment it is launched.

The program loads the script into memory line by line. During loading, the script is optimized and validated. Any syntax errors will be displayed, and they must be corrected before the script can run.

Script Startup (the Auto-execute Thread)

After the script has been loaded, the auto-execute thread begins executing at the script's top line, and continues until instructed to stop, such as by Return, ExitApp or Exit. The physical end of the script also acts as Exit.

The script will terminate after completing startup if it lacks hotkeys, hotstrings, visible GUIs, active timers, clipboard monitors and InputHooks, and has not called the Persistent function. Otherwise, it will stay running in an idle state, responding to events such as hotkeys, hotstrings, GUI events, custom menu items, and timers. If these conditions change after startup completes (for example,

the last timer is disabled), the script may exit when the last running thread completes or the last GUI closes.

Whenever any new thread is launched (whether by a hotkey, hotstring, timer, or some other event), the following settings are copied from the auto-execute thread. If not set by the auto-execute thread, the standard defaults will apply (as documented on each of the following pages): CoordMode, Critical, DetectHiddenText, DetectHiddenWindows, FileEncoding, ListLines, SendLevel, SendMode, SetControlDelay, SetDefaultMouseSpeed, SetKeyDelay, SetMouseDelay, SetRegView, SetStoreCapsLockMode, SetTitleMatchMode, SetWinDelay, and Thread.

Each thread retains its own collection of the above settings, so changes made to those settings will not affect other threads.

The "default setting" for one of the above functions usually refers to the current setting of the auto-execute thread, which starts out the same as the program-defined default setting.

Traditionally, the top of the script has been referred to as the auto-execute section. However, the auto-execute thread is not limited to just the top of the script. Any functions which are called on the auto-execute thread may also affect the default settings. As directives and function, hotkey, hotstring and class definitions are skipped when encountered during execution, it is possible for startup code to be placed throughout each file. For example, a global variable used by a group of hotkeys may be initialized above (or even below) those hotkeys rather than at the top of the script.

Splitting a Long Line into a Series of Shorter Ones

Long lines can be divided up into a collection of smaller ones to improve readability and maintainability. This does not reduce the script's execution speed because such lines are merged in memory the moment the script launches.

There are three methods, and they can generally be used in combination:

Continuation operator: Start or end a line with an expression operator to join it to the previous or next line.

Continuation by enclosure: A sub-expression enclosed in (), [] or {} can automatically span multiple lines in most cases.

Continuation section: Mark a group of lines to be merged together, with additional options such as what text (or code) to insert between lines.

Continuation operator: A line that starts with a comma or any other expression operator (except ++ and --) is automatically merged with the line directly above it. Similarly, a line that ends with an expression operator is automatically merged with the line below it. In the following example, the second line is appended to the first because it begins with a comma:

```
FileAppend "This is the text to append.`n"   ; A comment is allowed here.
   , A_ProgramFiles "\SomeApplication\LogFile.txt"  ; Comment.
```

Similarly, the following lines would get merged into a single line because the last two start with "and" or "or":

```
if Color = "Red" or Color = "Green"  or Color = "Blue"   ; Comment.
   or Color = "Black" or Color = "Gray" or Color = "White"   ; Comment.
   and ProductIsAvailableInColor(Product, Color)   ; Comment.
```

The ternary operator is also a good candidate:

```
ProductIsAvailable := (Color = "Red")
   ? false  ; We don't have any red products, so don't bother calling the function.
   : ProductIsAvailableInColor(Product, Color)
```

The following examples are equivalent to those above:

```
FileAppend "This is the text to append.`n",   ; A comment is allowed here.
   A_ProgramFiles "\SomeApplication\LogFile.txt"  ; Comment.

if Color = "Red" or Color = "Green"  or Color = "Blue" or   ; Comment.
```

```
    Color = "Black" or Color = "Gray" or Color = "White" and   ; Comment.
    ProductIsAvailableInColor(Product, Color)   ; Comment.

ProductIsAvailable := (Color = "Red") ?
    false : ; We don't have any red products, so don't bother calling the function.
    ProductIsAvailableInColor(Product, Color)
```

Although the indentation used in the examples above is optional, it might improve clarity by indicating which lines belong to ones above them. Also, blank lines or comments may be added between or at the end of any of the lines in the above examples.

A continuation operator cannot be used with an auto-replace hotstring or directive other than #HotIf.

Continuation by enclosure: If a line contains an expression or function/property definition with an unclosed (/[/{, it is joined with subsequent lines until the number of opening and closing symbols balances out. In other words, a sub-expression enclosed in parentheses, brackets or braces can automatically span multiple lines in most cases. For example:

```
myarray := [
  "item 1",
  "item 2",
]
MsgBox(
    "The value of item 2 is " myarray[2],
    "Title",
    "ok iconi"
    )
```

Continuation expressions may contain both types of comments.

Continuation expressions may contain normal continuation sections. Therefore, as with any line containing an expression, if a line begins with a non-escaped open parenthesis (), it is considered to be the start of a continuation section unless there is a closing parenthesis ()) on the same line.

Quoted strings cannot span multiple lines using this method alone. However, see above.

Continuation by enclosure can be combined with a continuation operator. For example:

```
myarray :=  ; The assignment operator causes continuation.
[  ; Brackets enclose the following two lines.
  "item 1",
  "item 2",
]
```

Brace ({) at the end of a line does not cause continuation if the program determines that it should be interpreted as the beginning of a block (OTB style) rather than the start of an object literal. Specifically (in descending order of precedence):

A brace is never interpreted as the beginning of a block if it is preceded by an unclosed (/[/{, since that would produce an invalid expression. For example, the brace in If ({ is the start of an object literal.

An object literal cannot legally follow) or], so if the brace follows either of those symbols (excluding whitespace), it is interpreted as the beginning of a block (such as for a function or property definition).

For control flow statements which require a body (and therefore support OTB), the brace can be the start of an object literal only if it is preceded by an operator, such as := { or for x in {. In particular, the brace in Loop { is always block-begin, and If { and While { are always errors.

A brace can be safely used for line continuation with any function call, expression or control flow statement which does not require a body. For example:

```
myfn() {
```

```
    return {
        key: "value"
    }
}
```

Continuation section: This method should be used to merge a large number of lines or when the lines are not suitable for the other methods. Although this method is especially useful for auto-replace hotstrings, it can also be used with any expression. For example:

```
; EXAMPLE #1:
Var := "
(
A line of text.
By default, the hard carriage return (Enter) between the previous line and this one will be stored.
        This line is indented with a tab; by default, that tab will also be stored.
Additionally, "quote marks" are automatically escaped when appropriate.
)"

; EXAMPLE #2:
FileAppend "
(
Line 1 of the text.
Line 2 of the text. By default, a linefeed (`n) is present between lines.
)", A_Desktop "\My File.txt"
```

In the examples above, a series of lines is bounded at the top and bottom by a pair of parentheses. This is known as a continuation section. Notice that any code after the closing parenthesis is also joined with the other lines (without any delimiter), but the opening and closing parentheses are not included.

If the line above the continuation section ends with a name character and the section does not start inside a quoted string, a single space is automatically inserted to separate the name from the contents of the continuation section.

Quote marks are automatically escaped (i.e. they are interpreted as literal characters) if the continuation section starts inside a quoted string, as in the examples above. Otherwise, quote marks act as they do normally; that is, continuation sections can contain expressions with quoted strings.

By default, leading spaces or tabs are omitted based on the indentation of the first line inside the continuation section. If the first line mixes spaces and tabs, only the first type of character is treated as indentation. If any line is indented less than the first line or with the wrong characters, all leading whitespace on that line is left as is.

The default behavior of a continuation section can be overridden by including one or more of the following options to the right of the section's opening parenthesis. If more than one option is present, separate each one from the previous with a space. For example: (LTrim Join|.

Join: Specifies how lines should be connected together. If this option is omitted, each line except the last will be followed by a linefeed character (`n). If the word Join is specified by itself, lines are connected directly to each other without any characters in between. Otherwise, the word Join should be followed immediately by as many as 15 characters. For example, Join`s would insert a space after each line except the last. Another example is Join`r`n, which inserts CR+LF between lines. Similarly, Join| inserts a pipe between lines. To have the final line in the section also ended by a join-string, include a blank line immediately above the section's closing parenthesis.

LTrim: Omits all spaces and tabs at the beginning of each line. This is usually unnecessary because of the default "smart" behaviour.

LTrim0 (LTrim followed by a zero): Turns off the omission of spaces and tabs from the beginning of each line.

RTrim0 (RTrim followed by a zero): Turns off the omission of spaces and tabs from the end of each line.

Comments (or Comment or Com or C): Allows semicolon comments inside the continuation section (but not /*..*/). Such comments (along with any spaces and tabs to their left) are entirely omitted from the joined result rather than being treated as literal text. Each comment can appear to the right of a line or on a new line by itself.

` (accent): Treats each backtick character literally rather than as an escape character. This also prevents the translation of any explicitly specified escape sequences such as `r and `t.

(or): If an opening or closing parenthesis appears to the right of the initial opening parenthesis (except as a parameter of the Join option), the line is reinterpreted as an expression instead of the beginning of a continuation section. This enables expressions like (x.y)[z]() to be used at the start of a line, and also allows multi-line expressions to start with a line like ((or (MyFunc(.

Escape sequences such as `n (linefeed) and `t (tab) are supported inside the continuation section except when the accent (`) option has been specified.

When the comment option is absent, semicolon and /*..*/ comments are not supported within the interior of a continuation section because they are seen as literal text. However, comments can be included on the bottom and top lines of the section. For example:

```
FileAppend "   ; Comment.
; Comment.
( LTrim Join    ; Comment.
    ; This is not a comment; it is literal. Include the word Comments in the line above to make it a comment.
)", "C:\File.txt"   ; Comment.
```

As a consequence of the above, semicolons never need to be escaped within a continuation section.

Since a closing parenthesis indicates the end of a continuation section, to have a line start with literal closing parenthesis, precede it with an accent/backtick: `). However, this cannot be combined with the accent (`) option.

A continuation section can be immediately followed by a line containing the open-parenthesis of another continuation section. This allows the options mentioned above to be varied during the course of building a single line.

The piecemeal construction of a continuation section by means of #Include is not supported.

Script Library Folders

The library folders provide a few standard locations to keep shared scripts which other scripts utilise by means of #Include. A library script typically contains a function or class which is designed to be used and reused in this manner. Placing library scripts in one of these locations makes it easier to write scripts that can be shared with others and work across multiple setups. The library locations are:

A_ScriptDir "\Lib\" ; Local library.

A_MyDocuments "\AutoHotkey\Lib\" ; User library.

"directory-of-the-currently-running-AutoHotkey.exe\Lib\" ; Standard library.

The library folders are searched in the order shown above.

For example, if a script includes the line #Include <MyLib>, the program searches for a file named "MyLib.ahk" in the local library. If not found there, it searches for it in the user library, and then the standard library. If a match is still not found and the library's name contains an underscore (e.g. MyPrefix_MyFunc), the program searches again with just the prefix (e.g. MyPrefix.ahk).

Although by convention a library file generally contains only a single function or class of the same name as its filename, it may also contain private functions that are called only by it. However, such functions should have fairly distinct names because they will still be in the global namespace; that is, they will be callable from anywhere in the script.

Convert a Script to an EXE (Ahk2Exe)

A script compiler (courtesy of fincs, with additions by TAC109) is included with the program.

Once a script is compiled, it becomes a standalone executable; that is, AutoHotkey.exe is not required in order to run the script. The compilation process creates an executable file which contains the following: the AutoHotkey interpreter, the script, any files it includes, and any files it has incorporated via the FileInstall function. Additional files can be included using compiler directives.

The same compiler is used for v1.1 and v2 scripts. The compiler distinguishes script versions by checking the major version of the base file supplied.

Compiler Topics

Running the Compiler

Base Executable File

Script Compiler Directives

Compressing Compiled Scripts

Background Information

Running the Compiler

Ahk2Exe can be used in the following ways:

GUI Interface: Run the "Convert .ahk to .exe" item in the Start Menu. (After invoking the GUI, there may be a pause before the window is shown; see Background Information for more details.)

Right-click: Within an open Explorer window, right-click any .ahk file and select "Compile Script" (only available if the script compiler option was chosen when AutoHotkey was installed). This creates an EXE file of the same base filename as the script, which appears after a short time in the same directory. Note: The EXE file is produced using the same custom icon, .bin file and compression setting that were last saved in Method #1 above, or as specified by any relevant compiler directive in the script.

Command Line: The compiler can be run from the command line by using the parameters shown below. If any command line parameters are used, the script is compiled immediately unless /gui is used. All parameters are optional, except that there must be one /gui or /in parameter.

Parameter pairMeaning

/in script_name The path and name of the script to compile. This is mandatory if any other parameters are used, unless /gui is used.

/out exe_name The path\name of the output .exe to be created. Default is the directory\base_name of the input file plus extension of .exe, or any relevant compiler directive in the script.

/icon icon_name The icon file to be used. Default is the last icon saved in the GUI interface, or any SetMainIcon compiler directive in the script.

/base file_name The base file to be used (a .bin or .exe file). The major version of the base file used must be the same as the version of the script to be compiled. Default is the last base file name saved in the GUI interface, or any Base compiler directive in the script.

/resourceid name Assigns a non-standard resource ID to be used for the main script for compilations which use an .exe base file (see Embedded Scripts). Numeric resource IDs should consist of a hash sign (#) followed by a decimal number. Default is #1, or any ResourceID compiler directive in the script.

/cp codepage Overrides the default codepage used to read script files. For a list of possible values, see Code Page Identifiers. Note that Unicode scripts should begin with a byte-order-mark (BOM), rendering the use of this parameter unnecessary.

/compress n Compress the exe? 0 = no, 1 = use MPRESS if present, 2 = use UPX if present. Default is the last setting saved in the GUI interface.

/gui Shows the GUI instead of immediately compiling. The other parameters can be used to override the settings last saved in the GUI. /in is optional in this case.

/silent [verbose] Disables all message boxes and instead outputs errors to the standard error stream (stderr); or to the standard output stream (stdout) if stderr fails. Other messages are also output to stdout. Optionally enter the word verbose to output status messages to stdout as well.

Deprecated:

/ahk file_name The path\name of AutoHotkey.exe to be used as a utility when compiling the script.

Deprecated:

/mpress 0or1 Compress the exe with MPRESS? 0 = no, 1 = yes. Default is the last setting used in the GUI interface.

Deprecated:

/bin file_name The .bin file to be used. Default is the last .bin file name saved in the GUI interface.

For example:

Ahk2Exe.exe /in "MyScript.ahk" /icon "MyIcon.ico"

Notes:

Parameters containing spaces must be enclosed in double quotes.

Compiling does not typically improve the performance of a script.

#NoTrayIcon and A_AllowMainWindow affect the behavior of compiled scripts.

The built-in variable A_IsCompiled contains 1 if the script is running in compiled form. Otherwise, it is blank.

When parameters are passed to Ahk2Exe, a message indicating the success or failure of the compiling process is written to stdout. Although the message will not appear at the command prompt, it can be "caught" by means such as redirecting output to a file.

Additionally in the case of a failure, Ahk2Exe has exit codes indicating the kind of error that occurred. These error codes can be found at GitHub (ErrorCodes.md).

The compiler's source code and newer versions can be found at GitHub.

Base Executable File

Each compiled script .exe is based on an executable file which implements the interpreter. The base files included in the Compiler directory have the ".bin" extension; these are versions of the interpreter which do not include the capability to load external script files. Instead, the program looks for a Win32 (RCDATA) resource named ">AUTOHOTKEY SCRIPT<" and loads that, or fails if it is not found.

The standard AutoHotkey executable files can also be used as the base of a compiled script, by embedding a Win32 (RCDATA) resource with ID 1. (Additional scripts can be added with the AddResource compiler directive.) This allows the compiled script .exe to be used with the /script switch to execute scripts other than the main embedded script. For more details, see Embedded Scripts.

Script Compiler Directives

Script compiler directives allow the user to specify details of how a script is to be compiled. Some of the features are:

Ability to change the version information (such as the name, description, version...).

Ability to add resources to the compiled script.

Ability to tweak several miscellaneous aspects of compilation.

Ability to remove code sections from the compiled script and vice versa.

See Script Compiler Directives for more details.

Compressing Compiled Scripts

Ahk2Exe optionally uses MPRESS or UPX freeware to compress compiled scripts. If MPRESS.exe and/or UPX.exe has been copied to the "Compiler" subfolder where AutoHotkey was installed, either can be used to compress the .exe as directed by the /compress parameter or the GUI setting.

MPRESS official website (downloads and information): http://www.matcode.com/mpress.htm

MPRESS mirror: https://www.autohotkey.com/mpress/

UPX official website (downloads and information): https://upx.github.io/

Note: While compressing the script executable prevents casual inspection of the script's source code using a plain text editor like Notepad or a PE resource editor, it does not prevent the source code from being extracted by tools dedicated to that purpose.

Background Information

The following folder structure is supported, where the running version of Ahk2Exe.exe is in the first \Compiler directory shown below:

```
\AutoHotkey
  AutoHotkeyA32.exe
  AutoHotkeyU32.exe
  AutoHotkeyU64.exe
  \Compiler
    Ahk2Exe.exe  ; the master version of Ahk2Exe
    ANSI 32-bit.bin
    Unicode 32-bit.bin
    Unicode 64-bit.bin
  \AutoHotkey v2.0-a135
    AutoHotkey32.exe
    AutoHotkey64.exe
    \Compiler
  \v2.0-beta.1
    AutoHotkey32.exe
    AutoHotkey64.exe
```

The base file search algorithm runs for a short amount of time when Ahk2Exe starts, and works as follows:

Qualifying AutoHotkey .exe files and all .bin files are searched for in the compiler's directory, the compiler's parent directory, and any of the compiler's sibling directories with directory names that start with AutoHotkey or V, but do not start with AutoHotkey_H. The selected directories are searched recursively. Any AutoHotkey.exe files found are excluded, leaving files such as AutoHotkeyA32.exe, AutoHotkey64.exe, etc. plus all .bin files found. All .exe files that are included must have a name starting with AutoHotkey and a file description containing the word AutoHotkey, and must have a version of 1.1.34+ or 2.0-a135+.

A version of the AutoHotkey interpreter is also needed (as a utility) for a successful compile, and one is selected using a similar algorithm. In most cases the version of the interpreter used will match the version of the base file selected by the user for the compile.

Passing Command Line Parameters to a Script

Scripts support command line parameters. The format is:

AutoHotkey.exe [Switches] [Script Filename] [Script Parameters]

And for compiled scripts, the format is:

CompiledScript.exe [Switches] [Script Parameters]

Switches: Zero or more of the following:

Switch Meaning Works compiled?

/force Launch unconditionally, skipping any warning dialogs. This has the same effect as #SingleInstance Off. Yes

/restart Indicate that the script is being restarted and should attempt to close a previous instance of the script (this is also used by the Reload function, internally). Yes

/ErrorStdOut

/ErrorStdOut=Encoding

Send syntax errors that prevent a script from launching to the standard error stream (stderr) rather than displaying a dialog. See #ErrorStdOut for details.

An encoding can optionally be specified. For example, /ErrorStdOut=UTF-8 encodes messages as UTF-8 before writing them to stderr.

No

/DebugConnect to a debugging client. For more details, see Interactive Debugging. No

/CPn

Overrides the default codepage used to read script files. For more details, see Script File Codepage.

No

/Validate

AutoHotkey loads the script and then exits instead of running it.

By default, load-time errors and warnings are displayed as usual. The /ErrorStdOut switch can be used to suppress or capture any error messages.

The process exit code is zero if the script successfully loaded, or non-zero if there was an error.

No

/iLib "OutFile"

Deprecated: Use /validate instead.

AutoHotkey loads the script but does not run it. In previous versions of AutoHotkey, filenames of auto-included files were written to the file specified by OutFile, formatted as #Include directives.

If the output file exists, it is overwritten. OutFile can be * to write the output to stdout.

If the script contains syntax errors, the output file may be empty. The process exit code can be used to detect this condition; if there is a syntax error, the exit code is 2. The /ErrorStdOut switch can be used to suppress or capture the error message.

No

/include "IncFile"

Includes a file prior to the main script. Only a single file can be included by this method. When the script is reloaded, this switch is automatically passed to the new instance.

No

/script

When used with a compiled script based on an .exe file, this switch causes the program to ignore the main embedded script. This allows a compiled script .exe to execute external script files or embedded scripts other than the main one. Other switches not normally supported by compiled scripts can be used but must be listed to the right of this switch. For example:

CompiledScript.exe /script /ErrorStdOut MyScript.ahk "Script's arg 1"

If the current executable file does not have an embedded script, this switch is permitted but has no effect.

This switch is not supported by compiled scripts which are based on a .bin file.

See also: Base Executable File (Ahk2Exe)

N/A

Script Filename: This can be omitted if there are no Script Parameters. If omitted, it defaults to the path and name of the AutoHotkey executable, replacing ".exe" with ".ahk". For example, if you rename AutoHotkey.exe to MyScript.exe, it will attempt to load MyScript.ahk. If you run AutoHotkey32.exe without parameters, it will attempt to load AutoHotkey32.ahk.

Specify an asterisk (*) for the filename to read the script text from standard input (stdin). This also puts the following into effect:

The initial working directory is used as A_ScriptDir and to locate the local Lib folder.

A_ScriptName and A_ScriptFullPath both contain "*".

#SingleInstance is off by default.

For an example, see ExecScript().

If the current executable file has embedded scripts, this parameter can be an asterisk followed by the resource name or ID of an embedded script. For compiled scripts (i.e. if an embedded script with the ID #1 exists), this parameter must be preceded by the /script switch.

Script Parameters: The string(s) you want to pass into the script, with each separated from the next by one or more spaces. Any parameter that contains spaces should be enclosed in quotation marks. If you want to pass an empty string as a parameter, specify two consecutive quotation marks. A literal quotation mark may be passed in by preceding it with a backslash (\"). Consequently, any trailing slash in a quoted parameter (such as "C:\My Documents\") is treated as a literal quotation mark (that is, the script would receive the string C:\My Documents"). To remove such quotes, use A_Args[1] := StrReplace(A_Args[1], '"')

Incoming parameters, if present, are stored as an array in the built-in variable A_Args, and can be accessed using array syntax. A_Args[1] contains the first parameter. The following example exits the script when too few parameters are passed to it:

```
if A_Args.Length < 3
{
   MsgBox "This script requires at least 3 parameters but it only received " A_Args.Length "."
   ExitApp
}
```

If the number of parameters passed into a script varies (perhaps due to the user dragging and dropping a set of files onto a script), the following example can be used to extract them one by one:

```
for n, param in A_Args  ; For each parameter:
{
   MsgBox "Parameter number " n " is " param "."
}
```

If the parameters are file names, the following example can be used to convert them to their case-corrected long names (as stored in the file system), including complete/absolute path:

```
for n, GivenPath in A_Args  ; For each parameter (or file dropped onto a script):
{
    Loop Files, GivenPath, "FD"  ; Include files and directories.
        LongPath := A_LoopFileFullPath
    MsgBox "The case-corrected long path name of file`n" GivenPath "`nis:`n" LongPath
}
```

Script File Codepage

In order for non-ASCII characters to be read correctly from file, the encoding used when the file was saved (typically by the text editor) must match what AutoHotkey uses when it reads the file. If it does not match, characters will be decoded incorrectly. AutoHotkey uses the following rules to decide which encoding to use:

If the file begins with a UTF-8 or UTF-16 (LE) byte order mark, the appropriate codepage is used and the /CPn switch is ignored.

If the /CPn switch is passed on the command-line, codepage n is used. For a list of possible values, see Code Page Identifiers.

In all other cases, UTF-8 is used (this default differs from AutoHotkey v1).

Note that this applies only to script files loaded by AutoHotkey, not to file I/O within the script itself. FileEncoding controls the default encoding of files read or written by the script, while IniRead and IniWrite always deal in UTF-16 or ANSI.

As all text is converted (where necessary) to the native string format, characters which are invalid or don't exist in the native codepage are replaced with a placeholder: '�'. This should only occur if there are encoding errors in the script file or the codepages used to save and load the file don't match.

RegWrite may be used to set the default for scripts launched from Explorer (e.g. by double-clicking a file):

```
; Uncomment the appropriate line below or leave them all commented to
;   reset to the default of the current build.  Modify as necessary:
; codepage := 0       ; System default ANSI codepage
; codepage := 65001    ; UTF-8
; codepage := 1200     ; UTF-16
; codepage := 1252     ; ANSI Latin 1; Western European (Windows)
if (codepage != "")
   codepage := " /CP" . codepage
cmd := Format('"{1}"{2} "%1" %*', A_AhkPath, codepage)
key := "AutoHotkeyScript\Shell\Open\Command"
if A_IsAdmin    ; Set for all users.
   RegWrite cmd, "REG_SZ", "HKCR\" key
else          ; Set for current user only.
   RegWrite cmd, "REG_SZ", "HKCU\Software\Classes\" key
```

This assumes AutoHotkey has already been installed. Results may be less than ideal if it has not.

Debugging a Script

Built-in functions such as ListVars and Pause can help you debug a script. For example, the following two lines, when temporarily inserted at carefully chosen positions, create "break points" in the script:

ListVars

Pause

When the script encounters these two lines, it will display the current contents of all variables for your inspection. When you're ready to resume, un-pause the script via the File or Tray menu. The script will then continue until reaching the next "break point" (if any).

It is generally best to insert these "break points" at positions where the active window does not matter to the script, such as immediately before a WinActivate function. This allows the script to properly resume operation when you un-pause it.

The following functions are also useful for debugging: ListLines, KeyHistory, and OutputDebug.

Some common errors, such as typos and missing "global" declarations, can be detected by enabling warnings.

Interactive Debugging

Interactive debugging is possible with a supported DBGp client. Typically the following actions are possible:

Set and remove breakpoints on lines - pause execution when a breakpoint is reached.

Step through code line by line - step into, over or out of functions.

Inspect all variables or a specific variable.

View the stack of running threads and functions.

Note that this functionality is disabled for compiled scripts which are based on a BIN file. For compiled scripts based on an EXE file, /debug must be specified after /script.

To enable interactive debugging, first launch a supported debugger client then launch the script with the /Debug command-line switch.

AutoHotkey.exe /Debug[=SERVER:PORT] ...

SERVER and PORT may be omitted. For example, the following are equivalent:

AutoHotkey /Debug "myscript.ahk"

AutoHotkey /Debug=localhost:9000 "myscript.ahk"

To attach the debugger to a script which is already running, send it a message as shown below:

```
ScriptPath := "" ; SET THIS TO THE FULL PATH OF THE SCRIPT
A_DetectHiddenWindows := true
if WinExist(ScriptPath " ahk_class AutoHotkey")
   ; Optional parameters:
   ;   wParam  = the IPv4 address of the debugger client, as a 32-bit integer.
   ;   lParam  = the port which the debugger client is listening on.
   PostMessage DllCall("RegisterWindowMessage",  "Str", "AHK_ATTACH_DEBUGGER")
```

Once the debugger client is connected, it may detach without terminating the script by sending the "detach" DBGp command.

Script Showcase

See this page for some useful scripts.

Variables and Expressions

Table of Contents

Variables

See Variables for general explanation and details about how variables work.

Storing values in variables: To store a string or number in a variable, use the colon-equal operator (:=) followed by a number, quoted string or any other type of expression. For example:

```
MyNumber := 123

MyString := "This is a literal string."

CopyOfVar := Var
```

A variable cannot be explicitly deleted, but its previous value can be released by assigning a new value, such as an empty string:

```
MyVar := ""
```

A variable can also be assigned a value indirectly, by taking its reference and using a double-deref or passing it to a function. For example:

```
MouseGetPos &x, &y
```

Reading the value of a variable which has not been assigned a value is considered an error. IsSet can be used to detect this condition.

Retrieving the contents of variables: To include the contents of a variable in a string, use concatenation or Format. For example:

```
MsgBox "The value of Var is " . Var . "."

MsgBox "The value in the variable named Var is " Var "."

MsgBox Format("Var has the value {1}.", Var)
```

Sub-expressions can be combined with strings in the same way. For example:

```
MsgBox("The sum of X and Y is " . (X + Y))
```

Comparing variables: Please read the expressions section below for important notes about the different kinds of comparisons.

Expressions

See Expressions for a structured overview and further explanation.

Expressions are used to perform one or more operations upon a series of variables, literal strings, and/or literal numbers.

Plain words in expressions are interpreted as variable names. Consequently, literal strings must be enclosed in double quotes to distinguish them from variables. For example:

```
if (CurrentSetting > 100 or FoundColor != "Blue")
    MsgBox "The setting is too high or the wrong color is present."
```

In the example above, "Blue" appears in quotes because it is a literal string. Single-quote marks (') and double-quote marks (") function identically, except that a string enclosed in single-quote marks can contain literal double-quote marks and vice versa. Therefore, to include an actual quote mark inside a literal string, escape the quote mark or enclose the string in the opposite type of quote mark. For example:

```
MsgBox "She said, `"An apple a day.`""
MsgBox 'She said, "An apple a day."'
```

Empty strings: To specify an empty string in an expression, use an empty pair of quotes. For example, the statement if (MyVar != "") would be true if MyVar is not blank.

Storing the result of an expression: To assign a result to a variable, use the := operator. For example:

```
NetPrice := Price * (1 - Discount/100)
```

Boolean values: When an expression is required to evaluate to true or false (such as an IF-statement), a blank or zero result is considered false and all other results are considered true. For example, the statement if ItemCount would be false only if ItemCount is blank or 0. Similarly, the expression if not ItemCount would yield the opposite result.

Operators such as NOT/>/=/< automatically produce a true or false value: they yield 1 for true and 0 for false. However, the AND/OR operators always produce one of the input values. For example, in the following expression, the variable Done is assigned 1 if A_Index is greater than 5 or the value of FoundIt in all other cases:

Done := A_Index > 5 or FoundIt

As hinted above, a variable can be used to hold a false value simply by making it blank or assigning 0 to it. To take advantage of this, the shorthand statement if Done can be used to check whether the variable Done is true or false.

In an expression, the keywords true and false resolve to 1 and 0. They can be used to make a script more readable as in these examples:

CaseSensitive := false

ContinueSearch := true

Integers and floating point: Within an expression, numbers are considered to be floating point if they contain a decimal point or scientific notation; otherwise, they are integers. For most operators -- such as addition and multiplication -- if either of the inputs is a floating point number, the result will also be a floating point number.

Within expressions and non-expressions alike, integers may be written in either hexadecimal or decimal format. Hexadecimal numbers all start with the prefix 0x. For example, Sleep 0xFF is equivalent to Sleep 255. Floating point numbers can optionally be written in scientific notation, with or without a decimal point (e.g. 1e4 or -2.1E-4).

Within expressions, unquoted literal numbers such as 128, 0x7F and 1.0 are converted to pure numbers before the script begins executing, so converting the number to a string may produce a value different to the original literal value. For example:

MsgBox(0x7F) ; Shows 128

MsgBox(1.00) ; Shows 1.0

Operators in Expressions

See Operators for general information about operators.

Except where noted below, any blank value (empty string) or non-numeric value involved in a math operation is not assumed to be zero. Instead, a TypeError is thrown. If Try is not used, the unhandled exception causes an error dialog by default.

Expression Operators (in descending precedence order)

Operator Description

%Expr%

Dereference or name substitution.

When Expr evaluates to a VarRef, %Expr% accesses the corresponding variable. For example, x := &y takes a reference to y and assigns it to x, then %x% := 1 assigns to the variable y and %x% reads its value.

Otherwise, the value of the sub-expression Expr is used as the name or partial name of a variable or property. This allows the script to refer to a variable or property whose name is determined by evaluating Expr, which is typically another variable. Variables cannot be created dynamically, but a variable can be assigned dynamically if it has been declared or referenced non-dynamically somewhere in the script.

Note: The result of the sub-expression Expr must be the name or partial name of the variable or property to be accessed.

Percent signs cannot be used directly within Expr due to ambiguity, but can be nested within parentheses. Otherwise, Expr can be any expression.

If there are any adjoining %Expr% sequences and partial names (without any spaces or other characters between them), they are combined to form a single name.

An Error is typically thrown if the variable does not already exist, or if it is uninitialized and its value is being read. The or-maybe operator (??) can be used to avoid that case by providing a default value. For example: %'novar'% ?? 42.

Although this is historically known as a "double-deref", this term is inaccurate when Expr does not contain a variable (first deref), and also when the resulting variable is the target of an assignment, not being dereferenced (second deref).

x.y

x.%z% Member access. Get or set a value or call a method of object x, where y is a literal name and z is an expression which evaluates to a name. See object syntax.

var?

Maybe. Permits the variable to be unset. This is valid only when passing a variable to an optional parameter, array element or object literal; or on the right-hand side of a direct assignment. The question mark must be followed by one of the following symbols (ignoring whitespace):)]},:. The variable may be passed conditionally via the ternary operator or on the right-hand side of AND/OR.

The variable is typically an optional parameter, but can be any variable. For variables that are not function parameters, a VarUnset warning may still be shown at load-time if there are other references to the variable but no assignments.

This operator is currently supported only for variables. To explicitly or conditionally omit a parameter in more general cases, use the unset keyword.

See also: unset (Optional Parameters)

++

--

Pre- and post-increment/decrement. Adds or subtracts 1 from a variable. The operator may appear either before or after the variable's name. If it appears before the name, the operation is performed and its result is used by the next operation (the result is a variable reference in this case). For example, Var := ++X increments X and then assigns its value to Var. Conversely, if the operator

appears after the variable's name, the result is the value of X prior to performing the operation. For example, Var := X++ increments X but Var receives the value X had before it was incremented.

These operators can also be used with a property of an object, such as myArray.Length++ or --myArray[i]. In these cases, the result of the sub-expression is always a number, not a variable reference.

**

Power. Example usage: base**exponent. Both base and exponent may contain a decimal point. If exponent is negative, the result will be formatted as a floating point number even if base and exponent are both integers. Since ** is of higher precedence than unary minus, -2**2 is evaluated like -(2**2) and so yields -4. Thus, to raise a literal negative number to a power, enclose it in parentheses such as (-2)**2.

The power operator is right-associative. For example, x ** y ** z is evaluated as x ** (y ** z).

Note: A negative base combined with a fractional exponent such as (-2)**0.5 is not supported; attempting it will cause an exception to be thrown. But both (-2)**2 and (-2)**2.0 are supported. If both base and exponent are 0, the result is undefined and an exception is thrown.

-

!

~

&

Unary minus (-): Inverts the sign of its operand.

Unary plus (+): +N is equivalent to -(-N). This has no effect when applied to a pure number, but can be used to convert numeric strings to pure numbers.

Logical-not (!): If the operand is blank or 0, the result of applying logical-not is 1, which means "true". Otherwise, the result is 0 (false). For example: !x or !(y and z). Note: The word NOT is

synonymous with ! except that ! has a higher precedence. Consecutive unary operators such as !!Var are allowed because they are evaluated in right-to-left order.

Bitwise-not (~): This inverts each bit of its operand. As 64-bit signed integers are used, a positive input value will always give a negative result and vice versa. For example, ~0xf0f evaluates to -0xf10 (-3856), which is binary equivalent to 0xfffffffffffff0f0. If an unsigned 32-bit value is intended, the result can be truncated with result & 0xffffffff. If the operand is a floating point value, a TypeError is thrown.

Reference (&): Creates a VarRef, which is a value representing a reference to a variable. A VarRef can then be used to indirectly access the target variable. For example, ref := &target followed by %ref% := 1 would assign the value 1 to target. The VarRef would typically be passed to a function, but could be stored in an array or property. See also: Dereference, ByRef.

Taking a reference to a built-in variable such as A_Clipboard is currently not supported, except when being passed directly to an OutputVar parameter of a built-in function.

*

/

//

Multiply (*): The result is an integer if both inputs are integers; otherwise, it is a floating point number.

Other uses: The asterisk (*) symbol is also used in variadic function calls.

True divide (/): True division yields a floating point result even when both inputs are integers. For example, 3/2 yields 1.5 rather than 1, and 4/2 yields 2.0 rather than 2.

Integer divide (//): The double-slash operator uses high-performance integer division. For example, 5//3 is 1 and 5//-3 is -1. If either of the inputs is in floating point format, a TypeError is thrown. For modulo, see Mod.

The *= and /= operators are a shorthand way to multiply or divide the value in a variable by another value. For example, Var*=2 produces the same result as Var:=Var*2 (though the former performs better).

Division by zero causes a ZeroDivisionError to be thrown.

+

-

Add (+) and subtract (-). On a related note, the += and -= operators are a shorthand way to increment or decrement a variable. For example, Var+=2 produces the same result as Var:=Var+2 (though the former performs better). Similarly, a variable can be increased or decreased by 1 by using Var++, Var--, ++Var, or --Var.

Other uses: If the + or - symbol is not preceded by a value (or a sub-expression which yields a value), it is interpreted as a unary operator instead.

<<

>>

>>>

Bit shift left (<<). Example usage: Value1 << Value2. This is equivalent to multiplying Value1 by "2 to the Value2th power".

Arithmetic bit shift right (>>). Example usage: Value1 >> Value2. This is equivalent to dividing Value1 by "2 to the Value2th power" and rounding the result to the nearest integer leftward on the number line; for example, -3>>1 is -2.

Logical bit shift right (>>>). Example usage: Value1 >>> Value2. Unlike arithmetic bit shift right, this does not preserve the sign of the number. For example, -1 has the same bit representation as the unsigned 64-bit integer 0xffffffffffffffff, therefore -1 >>> 1 is 0x7fffffffffffffff.

The following applies to all three operators:

If either of the inputs is a floating-point number, a TypeError is thrown.

If Value2 is less than 0 or greater than 63, an exception is thrown.

&

^

|

Bitwise-and (&), bitwise-exclusive-or (^), and bitwise-or (|). Of the three, & has the highest precedence and | has the lowest.

If either of the inputs is a floating-point number, a TypeError is thrown.

Related: Bitwise-not (~)

.

Concatenate. A period (dot) with at least one space or tab on each side is used to combine two items into a single string. You may also omit the period to achieve the same result (except where ambiguous such as x -y, or when the item on the right side has a leading ++ or --). When the dot is omitted, there must be at least one space or tab between the items to be merged.

Var := "The color is " . FoundColor ; Explicit concat

Var := "The color is " FoundColor ; Auto-concat

Sub-expressions can also be concatenated. For example: Var := "The net price is " . Price * (1 - Discount/100).

A line that begins with a period (or any other operator) is automatically appended to the line above it.

The entire length of each input is used, even if it includes binary zero. For example, Chr(0x2010) Chr(0x0000) Chr(0x4030) produces the following string of bytes (due to UTF-16-LE encoding): 0x10, 0x20, 0, 0, 0x30, 0x40. The result has an additional null-terminator (binary zero) which is not included in the length.

Other uses: If there is no space or tab to the right of a period (dot), it is interpreted as either a literal floating-point number or member access. For example, 1.1 and (.5) are numbers, A_Args.Has(3) is a method call and A_Args.Length is a property access.

~= Shorthand for RegExMatch. For example, the result of "abc123" ~= "\d" is 4 (the position of the first numeric character).

> <

>= <=

Greater (>), less (<), greater-or-equal (>=), and less-or-equal (<=). The inputs are compared numerically. A TypeError is thrown if either of the inputs is not a number or a numeric string.

=

==

!=

!==

Case-insensitive equal (=) / not-equal (!=) and case-sensitive equal (==) / not-equal (!==). The == operator behaves identically to = except when either of the inputs is not numeric (or both are strings), in which case == is always case sensitive and = is always case insensitive. The != and !== behave identically to their counterparts without !, except that the result is inverted.

The == and !== operators can be used to compare strings which contain binary zero. All other comparison operators except ~= compare only up to the first binary zero.

For case insensitive comparisons, only the ASCII letters A-Z are considered equivalent to their lowercase counterparts. To instead compare according to the rules of the current user's locale, use StrCompare and specify "Locale" for the CaseSense parameter.

IS

IN

CONTAINS

Value is Class yields true (1) if Value is an instance of Class or false (0) otherwise. Class must be an Object with a Prototype own property, but typically the property is defined implicitly by a class definition. This operation is generally equivalent to HasBase(Value, Class.Prototype).

in and contains are reserved for future use.

NOT Logical-NOT. Except for its lower precedence, this is the same as the ! operator. For example, not (x = 3 or y = 3) is the same as !(x = 3 or y = 3).

AND

&&

Both of these are logical-AND. For example: x > 3 and x < 10.

In an expression where all operands resolve to True, the last operand that resolved to True is returned. Otherwise, the first operand that resolves to False is returned. Effectively, only when all operands are true, will the result be true. Boolean expressions are subject to short-circuit evaluation (left to right) in order to enhance performance.

A := 1, B := {}, C := 20, D := True, E := "String" ; All operands are truthy and will be evaluated

MsgBox(A && B && C && D && E) ; The last truthy operand is returned ("String")

A := 1, B := "", C := 0, D := False, E := "String" ; B is falsey, C and D are false

MsgBox(A && B && ++C && D && E) ; The first falsey operand is returned (""). C, D and E are not evaluated and C is never incremented

A line that begins with AND or && (or any other operator) is automatically appended to the line above it.

OR

||

Both of these are logical-OR. For example: x <= 3 or x >= 10.

In an expression where at least one operand resolves to True, the first operand that resolved to True is returned. Otherwise, the last operand that resolves to False is returned. Effectively, provided at least one operand is true, the result will be true. Boolean expressions are subject to short-circuit evaluation (left to right) in order to enhance performance.

A := "", B := False, C := 0, D := "String", E := 20 ; At least one operand is truthy. All operands up until D (including) will be evaluated

MsgBox(A || B || C || D || ++E) ; The first truthy operand is returned ("String"). E is not evaluated and is never incremented

A := "", B := False, C := 0 ; All operands are falsey and will be evaluated

MsgBox(A || B || C) ; The last falsey operand is returned (0)

A line that begins with OR or || (or any other operator) is automatically appended to the line above it.

??

Or maybe, otherwise known as the coalescing operator. If the left operand (which must be a variable) has a value, it becomes the result and the right branch is skipped. Otherwise, the right operand becomes the result. In other words, A ?? B behaves like A || B (logical-OR) except that the condition is IsSet(A).

This is typically used to provide a default value when it is known that a variable or optional parameter might not already have a value. For example:

MsgBox MyVar ?? "Default value"

Since the variable is expected to sometimes be uninitialized, no error is thrown in that case. Unlike IsSet(A) ? A : B, a VarUnset warning may still be shown at load-time if there are other references to the variable but no assignments.

?:

Ternary operator. This operator is a shorthand replacement for the if-else statement. It evaluates the condition on its left side to determine which of its two branches should become its final result. For example, var := x>y ? 2 : 3 stores 2 in Var if x is greater than y; otherwise it stores 3. To enhance performance, only the winning branch is evaluated (see short-circuit evaluation).

See also: maybe (var?), or-maybe (??)

Note: When used at the beginning of a line, the ternary condition should usually be enclosed in parentheses to reduce ambiguity with other types of statements. For details, see Expression Statements.

:=

+=

-=

*=

/=

//=

.=

|=

&=

^=

>>=

<<=

>>>=

Assign. Performs an operation on the contents of a variable and stores the result back in the same variable. The simplest assignment operator is colon-equals (:=), which stores the result of an expression in a variable. For a description of what the other operators do, see their related entries

in this table. For example, Var //= 2 performs integer division to divide Var by 2, then stores the result back in Var. Similarly, Var .= "abc" is a shorthand way of writing Var := Var . "abc".

Unlike most other operators, assignments are evaluated from right to left. Consequently, a line such as Var1 := Var2 := 0 first assigns 0 to Var2 then assigns Var2 to Var1.

If an assignment is used as the input for some other operator, its value is the variable itself. For example, the expression (Var+=2) > 50 is true if the newly-increased value in Var is greater than 50. It is also valid to combine an assignment with the reference operator, as in &(Var := "initial value").

The precedence of the assignment operators is automatically raised when it would avoid a syntax error or provide more intuitive behavior. For example: not x:=y is evaluated as not (x:=y). Similarly, ++Var := X is evaluated as ++(Var := X); and Z>0 ? X:=2 : Y:=2 is evaluated as Z>0 ? (X:=2) : (Y:=2).

The target variable can be un-set by combining a direct assignment (:=) with the unset keyword or the maybe (var?) operator. For example: Var := unset, Var1 := (Var2?).

An assignment can also target a property of an object, such as myArray.Length += n or myArray[i] .= t. When assigning to a property, the result of the sub-expression is the value being assigned, not a variable reference.

() => expr

Fat arrow function. Defines a simple function and returns a Func or Closure object. Write the function's parameter list (optionally preceded by a function name) to the left of the operator. When the function is called (via the returned reference), it evaluates the sub-expression expr and returns the result.

The following two examples are equivalent:

sumfn := Sum(a, b) => a + b

```
Sum(a, b) {
   return a + b
}
sumfn := Sum
```

In both cases, the function is defined unconditionally at the moment the script launches, but the function reference is stored in sumfn only if and when the assignment is evaluated.

If the function name is omitted and the parameter list consists of only a single parameter name, the parentheses can be omitted. The example below defines an anonymous function with one parameter a and stores its reference in the variable double:

```
double := a => a * 2
```

Variable references in expr are resolved in the same way as in the equivalent full function definition. For instance, expr may refer to an outer function's local variables (as in any nested function), in which case a new closure is created and returned each time the fat arrow expression is evaluated. The function is always assume-local, since declarations cannot be used.

Specifying a name for the function allows it to be called recursively or by other nested functions without storing a reference to the closure within itself (thereby creating a problematic circular reference). It can also be helpful for debugging, such with Func.Name or when displayed on the debugger's call stack.

Fat arrow syntax can also be used to define shorthand properties and methods.

,

Comma (multi-statement). Commas may be used to write multiple sub-expressions on a single line. This is most commonly used to group together multiple assignments or function calls. For example: x:=1, y+=2, ++index, MyFunc(). Such statements are executed in order from left to right.

Note: A line that begins with a comma (or any other operator) is automatically appended to the line above it. See also: comma performance.

Comma is also used to delimit the parameters of a function call or control flow statement. To include a multi-statement expression in a parameter list, enclose it in an extra set of parentheses. For example, MyFn((x, y)) evaluates both x and y but passes y as the first and only parameter of MyFn.

The following types of sub-expressions override precedence/order of evaluation:

Expression Description

(expression)

Any sub-expression enclosed in parentheses. For example, (3 + 2) * 2 forces 3 + 2 to be evaluated first.

For a multi-statement expression, the result of the last statement is returned. For example, (a := 1, b := 2, c := 3) returns 3.

Mod()

Round()

Abs()

Function call. There must be no space between the function name or expression and the open parenthesis which begins the parameter list. For details, see Function Calls.

(expression) is not necessarily required to be enclosed in parentheses, but doing so may eliminate ambiguity. For example, (x.y)() retrieves a function from a property and then calls it with no parameters, whereas x.y() would implicitly pass x as the first parameter.

(expression)()

Fn(Params*)

Variadic function call. Params is an enumerable object (an object with an __Enum method), such as an Array containing parameter values.

x[y]

[a, b, c]

Item access. Get or set the __Item property (or default property) of object x with the parameter y (or multiple parameters in place of y). This typically corresponds to an array element or item within a collection, where y is the item's index or key. The item can be assigned a value by using any assignment operator immediately after the closing bracket. For example, x[y] := z.

Array literal. If the open-bracket is not preceded by a value (or a sub-expression which yields a value), it is interpreted as the beginning of an array literal. For example, [a, b, c] is equivalent to Array(a, b, c) (a, b and c are variables).

See Arrays and Maps for common usage.

{a: b, c: d}

Object literal. Create an Object. Each pair consists of a literal property name a and a property value expression b. For example, x := {a: b} is equivalent to x := Object(), x.a := b. Base may be set within the object literal, but all other properties are set as own value properties, potentially overriding properties inherited from the base object.

To use a dynamic property name, enclose the sub-expression in percent signs. For example: {%nameVar%: valueVar}.

Performance: The comma operator is usually faster than writing separate expressions, especially when assigning one variable to another (e.g. x:=y, a:=b). Performance continues to improve as more and more expressions are combined into a single expression; for example, it may be 35% faster to combine five or ten simple expressions into a single expression.

Built-in Variables

The variables below are built into the program and can be referenced by any script.

See Built-in Variables for general information.

Table of Contents

Special Characters

Variable	Description
A_Space	Contains a single space character.
A_Tab	Contains a single tab character.

Script Properties

Variable	Description
A_Args	Contains an array of command line parameters. For details, see Passing Command Line Parameters to a Script.
A_WorkingDir	

Can be used to get or set the script's current working directory, which is where files will be accessed by default. The final backslash is not included unless it is the root directory. Two examples: C:\ and C:\My Documents.

Use SetWorkingDir or assign a path to A_WorkingDir to change the working directory.

The script's working directory defaults to A_ScriptDir, regardless of how the script was launched.

A_InitialWorkingDir The script's initial working directory, which is determined by how it was launched. For example, if it was run via shortcut -- such as on the Start Menu -- its initial working directory is determined by the "Start in" field within the shortcut's properties.

A_ScriptDir

The full path of the directory where the current script is located. The final backslash is omitted (even for root directories).

If the script text is read from stdin rather than from file, this variable contains the initial working directory.

A_ScriptName

Can be used to get or set the default title for MsgBox, InputBox, FileSelect, DirSelect and Gui. If not set by the script, it defaults to the file name of the current script, without its path, e.g. MyScript.ahk.

If the script text is read from stdin rather than from file, the default value is "*".

If the script is compiled or embedded, this is the name of the current executable file.

A_ScriptFullPath

The full path of the current script, e.g. C:\Scripts\My Script.ahk

If the script text is read from stdin rather than from file, the value is "*".

If the script is compiled or embedded, this is the full path of the current executable file.

A_ScriptHwnd The unique ID (HWND/handle) of the script's hidden main window.

A_LineNumber

The number of the currently executing line within the script (or one of its #Include files). This line number will match the one shown by ListLines; it can be useful for error reporting such as this example: MsgBox "Could not write to log file (line number " A_LineNumber ")".

Since a compiled script has merged all its #Include files into one big script, its line numbering may be different than when it is run in non-compiled mode.

A_LineFile

The full path and name of the file to which A_LineNumber belongs. If the script was loaded from an external file, this is the same as A_ScriptFullPath unless the line belongs to one of the script's #Include files.

If the script was compiled based on a .bin file, this is the full path and name of the current executable file, the same as A_ScriptFullPath.

If the script is embedded, A_LineFile contains an asterisk (*) followed by the resource name; e.g. *#1

A_ThisFunc The name of the user-defined function that is currently executing (blank if none); for example: MyFunction. See also: Name property (Func)

A_AhkVersion Contains the version of AutoHotkey that is running the script, such as 1.0.22. In the case of a compiled script, the version that was originally used to compile it is reported. The formatting of the version number allows a script to check whether A_AhkVersion is greater than some minimum version number with > or >= as in this example: if (A_AhkVersion >= "1.0.25.07"). See also: #Requires and VerCompare

A_AhkPath

For non-compiled or embedded scripts: The full path and name of the EXE file that is actually running the current script. For example: C:\Program Files\AutoHotkey\AutoHotkey.exe

For compiled scripts based on a .bin file, the value is determined by reading the installation directory from the registry and appending "\AutoHotkey.exe". If AutoHotkey is not installed, the value is blank. The example below is equivalent:

InstallDir := RegRead("HKLM\SOFTWARE\AutoHotkey", "InstallDir", "")

AhkPath := InstallDir ? InstallDir "\AutoHotkey.exe" : ""

For compiled scripts based on an .exe file, A_AhkPath contains the full path of the compiled script. This can be used in combination with /script to execute external scripts. To instead locate the installed copy of AutoHotkey, read the registry as shown above.

A_IsCompiled Contains 1 if the script is running as a compiled EXE and an empty string (which is considered false) if it is not.

Date and Time

Variable Description

A_YYYY

Current 4-digit year (e.g. 2004). Synonymous with A_Year.

Note: To retrieve a formatted time or date appropriate for your locale and language, use FormatTime() (time and long date) or FormatTime(, "LongDate") (retrieves long-format date).

A_MM Current 2-digit month (01-12). Synonymous with A_Mon.

A_DD Current 2-digit day of the month (01-31). Synonymous with A_MDay.

A_MMMM Current month's full name in the current user's language, e.g. July

A_MMM Current month's abbreviation in the current user's language, e.g. Jul

A_DDDD Current day of the week's full name in the current user's language, e.g. Sunday

A_DDD Current day of the week's abbreviation in the current user's language, e.g. Sun

A_WDay Current 1-digit day of the week (1-7). 1 is Sunday in all locales.

A_YDay Current day of the year (1-366). The value is not zero-padded, e.g. 9 is retrieved, not 009. To retrieve a zero-padded value, use the following: FormatTime(, "YDay0").

A_YWeek Current year and week number (e.g. 200453) according to ISO 8601. To separate the year from the week, use Year := SubStr(A_YWeek, 1, 4) and Week := SubStr(A_YWeek, -2). Precise definition of A_YWeek: If the week containing January 1st has four or more days in the new year, it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1.

A_Hour Current 2-digit hour (00-23) in 24-hour time (for example, 17 is 5pm). To retrieve 12-hour time as well as an AM/PM indicator, follow this example: FormatTime(, "h:mm:ss tt")

A_Min

Current 2-digit minute (00-59).

A_Sec Current 2-digit second (00-59).

A_MSec Current 3-digit millisecond (000-999). To remove the leading zeros, follow this example: Milliseconds := A_MSec + 0.

A_Now

The current local time in YYYYMMDDHH24MISS format.

Note: Date and time math can be performed with DateAdd and DateDiff. Also, FormatTime can format the date and/or time according to your locale or preferences.

A_NowUTC The current Coordinated Universal Time (UTC) in YYYYMMDDHH24MISS format. UTC is essentially the same as Greenwich Mean Time (GMT).

A_TickCount

The number of milliseconds that have elapsed since the system was started, up to 49.7 days. By storing A_TickCount in a variable, elapsed time can later be measured by subtracting that variable from the latest A_TickCount value. For example:

StartTime := A_TickCount

Sleep 1000

ElapsedTime := A_TickCount - StartTime

MsgBox ElapsedTime " milliseconds have elapsed."

If you need more precision than A_TickCount's 10ms, use QueryPerformanceCounter().

Script Settings

Variable Description

A_IsSuspended Contains 1 if the script is suspended and 0 otherwise.

A_IsPaused Contains 1 if the thread immediately underneath the current thread is paused. Otherwise it contains 0.

A_IsCritical Contains 0 if Critical is off for the current thread. Otherwise it contains an integer greater than zero, namely the message-check frequency being used by Critical. The current state of Critical can be saved and restored via Old_IsCritical := A_IsCritical followed later by A_IsCritical := Old_IsCritical.

A_ListLines Can be used to get or set whether to log lines. Possible values are 0 (disabled) and 1 (enabled). For details, see ListLines.

A_TitleMatchMode Can be used to get or set the title match mode. Possible values are 1, 2, 3 and RegEx. For details, see SetTitleMatchMode.

A_TitleMatchModeSpeed Can be used to get or set the title match speed. Possible values are Fast and Slow. For details, see SetTitleMatchMode.

A_DetectHiddenWindows Can be used to get or set whether to detect hidden windows. Possible values are 0 (disabled) and 1 (enabled). For details, see DetectHiddenWindows.

A_DetectHiddenText Can be used to get or set whether to detect hidden text in a window. Possible values are 0 (disabled) and 1 (enabled). For details, see DetectHiddenText.

A_FileEncoding Can be used to get or set the default encoding for various built-in functions. For details, see FileEncoding.

A_SendMode Can be used to get or set the send mode. Possible values are Event, Input, Play and InputThenPlay. For details, see SendMode.

A_SendLevel Can be used to get or set the send level, an integer from 0 to 100. For details, see SendLevel.

A_StoreCapsLockMode Can be used to get or set whether to restore the state of CapsLock after a Send. Possible values are 0 (disabled) and 1 (enabled). For details, see SetStoreCapsLockMode.

A_KeyDelay

A_KeyDuration Can be used to get or set the delay or duration for keystrokes, in milliseconds. For details, see SetKeyDelay.

A_KeyDelayPlay

A_KeyDurationPlay Can be used to get or set the delay or duration for keystrokes sent via SendPlay mode, in milliseconds. For details, see SetKeyDelay.

A_WinDelay Can be used to get or set the delay for windowing functions, in milliseconds. For details, see SetWinDelay.

A_ControlDelay Can be used to get or set the delay for control-modifying functions, in milliseconds. For details, see SetControlDelay.

A_MenuMaskKey Controls which key is used to mask Win or Alt keyup events. For details, see A_MenuMaskKey.

A_MouseDelay

A_MouseDelayPlay Can be used to get or set the mouse delay, in milliseconds. A_MouseDelay is for the traditional SendEvent mode, whereas A_MouseDelayPlay is for SendPlay. For details, see SetMouseDelay.

A_DefaultMouseSpeed Can be used to get or set the default mouse speed, an integer from 0 (fastest) to 100 (slowest). For details, see SetDefaultMouseSpeed.

A_CoordModeToolTip

A_CoordModePixel

A_CoordModeMouse

A_CoordModeCaret

A_CoordModeMenu Can be used to get or set the area to which coordinates are to be relative. Possible values are Screen, Window and Client. For details, see CoordMode.

A_RegView Can be used to get or set the registry view. Possible values are 32, 64 and Default. For details, see SetRegView.

A_TrayMenu

Returns a Menu object which can be used to modify or display the tray menu.

A_AllowMainWindow

Can be used to get or set whether the script's main window is allowed to be opened via the tray icon. Possible values are 0 (forbidden) and 1 (allowed).

If the script is neither compiled nor embedded, this variable defaults to 1, otherwise this variable defaults to 0 but can be overridden by assigning it a value. Setting it to 1 also restores the "Open" option to the tray menu and enables the items in the main window's View menu such as "Lines most recently executed", which allows viewing of the script's source code and other info.

The following functions are always able to show the main window and activate the corresponding View options when they are encountered in the script at runtime: ListLines, ListVars, ListHotkeys, and KeyHistory.

Setting it to 1 does not prevent the main window from being shown by WinShow or inspected by ControlGetText or similar methods, but it does prevent the script's source code and other info from being exposed via the main window, except when one of the functions listed above is called by the script.

A_IconHidden Can be used to get or set whether to hide the tray icon. Possible values are 0 (visible) and 1 (hidden). For details, see #NoTrayIcon.

A_IconTip

Can be used to get or set the tray icon's tooltip text, which is displayed when the mouse hovers over it. If blank, the script's name is used instead.

To create a multi-line tooltip, use the linefeed character (`n) in between each line, e.g. "Line1`nLine2". Only the first 127 characters are displayed, and the text is truncated at the first tab character, if present.

On Windows 10 and earlier, to display tooltip text containing ampersands (&), escape each ampersand with two additional ampersands. For example, assigning "A &&& B" would display "A & B" in the tooltip.

A_IconFile Blank unless a custom tray icon has been specified via TraySetIcon -- in which case it is the full path and name of the icon's file.

A_IconNumber Blank if A_IconFile is blank. Otherwise, it's the number of the icon in A_IconFile (typically 1).

User Idle Time

Variable Description

A_TimeIdle The number of milliseconds that have elapsed since the system last received keyboard, mouse, or other input. This is useful for determining whether the user is away. Physical input from the user as well as artificial input generated by any program or script (such as the Send or MouseMove functions) will reset this value back to zero. Since this value tends to increase by increments of 10, do not check whether it is equal to another value. Instead, check whether it is greater or less than another value. For example:

if A_TimeIdle > 600000

MsgBox "The last keyboard or mouse activity was at least 10 minutes ago."

A_TimeIdlePhysical Similar to above but ignores artificial keystrokes and/or mouse clicks whenever the corresponding hook (keyboard or mouse) is installed; that is, it responds only to physical events. (This prevents simulated keystrokes and mouse clicks from falsely indicating that a user is present.) If neither hook is installed, this variable is equivalent to A_TimeIdle. If only one hook is installed, only its type of physical input affects A_TimeIdlePhysical (the other/non-installed hook's input, both physical and artificial, has no effect).

A_TimeIdleKeyboard If the keyboard hook is installed, this is the number of milliseconds that have elapsed since the system last received physical keyboard input. Otherwise, this variable is equivalent to A_TimeIdle.

A_TimeIdleMouse If the mouse hook is installed, this is the number of milliseconds that have elapsed since the system last received physical mouse input. Otherwise, this variable is equivalent to A_TimeIdle.

Hotkeys, Hotstrings, and Custom Menu Items

Variable Description

A_ThisHotkey

The most recently executed hotkey or non-auto-replace hotstring (blank if none), e.g. #z. This value will change if the current thread is interrupted by another hotkey or hotstring, so it is generally better to use the parameter ThisHotkey when available.

When a hotkey is first created -- either by the Hotkey function or the double-colon syntax in the script -- its key name and the ordering of its modifier symbols becomes the permanent name of that hotkey, shared by all variants of the hotkey.

When a hotstring is first created, the exact text used to create it becomes the permanent name of the hotstring.

A_PriorHotkey Same as above except for the previous hotkey. It will be blank if none.

A_PriorKey The name of the last key which was pressed prior to the most recent key-press or key-release, or blank if no applicable key-press can be found in the key history. All input generated by AutoHotkey scripts is excluded. For this variable to be of use, the keyboard or mouse hook must be installed and key history must be enabled.

A_TimeSinceThisHotkey The number of milliseconds that have elapsed since A_ThisHotkey was pressed. It will be blank whenever A_ThisHotkey is blank.

A_TimeSincePriorHotkey The number of milliseconds that have elapsed since A_PriorHotkey was pressed. It will be blank whenever A_PriorHotkey is blank.

A_EndChar The ending character that was pressed by the user to trigger the most recent non-auto-replace hotstring. If no ending character was required (due to the * option), this variable will be blank.

A_MaxHotkeysPerInterval The maximum number of hotkeys that can be pressed within the interval defined by A_HotkeyInterval without triggering a warning dialog. For details, see A_MaxHotkeysPerInterval.

A_HotkeyInterval The length of the interval used by A_MaxHotkeysPerInterval, in milliseconds.

A_HotkeyModifierTimeout Affects the behavior of Send with hotkey modifiers: Ctrl, Alt, Win, and Shift. For details, see A_HotkeyModifierTimeout.

Operating System and User Info

Variable Description

A_ComSpec

Contains the same string as the environment's ComSpec variable. Often used with Run/RunWait. For example:

C:\Windows\system32\cmd.exe

A_Temp

The full path and name of the folder designated to hold temporary files. It is retrieved from one of the following locations (in order): 1) the environment variables TMP, TEMP, or USERPROFILE; 2) the Windows directory. For example:

C:\Users\<UserName>\AppData\Local\Temp

A_OSVersion

The version number of the operating system, in the format "major.minor.build". For example, Windows 7 SP1 is 6.1.7601.

Applying compatibility settings in the AutoHotkey executable or compiled script's properties causes the OS to report a different version number, which is reflected by A_OSVersion.

A_Is64bitOS Contains 1 (true) if the OS is 64-bit or 0 (false) if it is 32-bit.

A_PtrSize Contains the size of a pointer, in bytes. This is either 4 (32-bit) or 8 (64-bit), depending on what type of executable (EXE) is running the script.

A_Language The system's default language, which is one of these 4-digit codes.

A_ComputerName The name of the computer as seen on the network.

A_UserName The logon name of the user who launched this script.

A_WinDir The Windows directory. For example: C:\Windows

A_ProgramFiles

The Program Files directory (e.g. C:\Program Files or C:\Program Files (x86)). This is usually the same as the ProgramFiles environment variable.

On 64-bit systems (and not 32-bit systems), the following applies:

If the executable (EXE) that is running the script is 32-bit, A_ProgramFiles returns the path of the "Program Files (x86)" directory.

For 32-bit processes, the ProgramW6432 environment variable contains the path of the 64-bit Program Files directory. On Windows 7 and later, it is also set for 64-bit processes.

The ProgramFiles(x86) environment variable contains the path of the 32-bit Program Files directory.

A_AppData

The full path and name of the folder containing the current user's application-specific data. For example:

C:\Users\<UserName>\AppData\Roaming

A_AppDataCommon

The full path and name of the folder containing the all-users application-specific data. For example:

C:\ProgramData

A_Desktop

The full path and name of the folder containing the current user's desktop files. For example:

C:\Users\<UserName>\Desktop

A_DesktopCommon

The full path and name of the folder containing the all-users desktop files. For example:

C:\Users\Public\Desktop

A_StartMenu

The full path and name of the current user's Start Menu folder. For example:

C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\Start Menu

A_StartMenuCommon

The full path and name of the all-users Start Menu folder. For example:

C:\ProgramData\Microsoft\Windows\Start Menu

A_Programs

The full path and name of the Programs folder in the current user's Start Menu. For example:

C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs

A_ProgramsCommon

The full path and name of the Programs folder in the all-users Start Menu. For example:

C:\ProgramData\Microsoft\Windows\Start Menu\Programs

A_Startup

The full path and name of the Startup folder in the current user's Start Menu. For example:

C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup

A_StartupCommon

The full path and name of the Startup folder in the all-users Start Menu. For example:

C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup

A_MyDocuments

The full path and name of the current user's "My Documents" folder. Unlike most of the similar variables, if the folder is the root of a drive, the final backslash is not included (e.g. it would contain M: rather than M:\). For example:

C:\Users\<UserName>\Documents

A_IsAdmin

If the current user has admin rights, this variable contains 1. Otherwise, it contains 0.

To have the script restart itself as admin (or show a prompt to the user requesting admin), use Run *RunAs. However, note that running the script as admin causes all programs launched by the script to also run as admin. For a possible alternative, see the FAQ.

A_ScreenWidth

A_ScreenHeight

The width and height of the primary monitor, in pixels (e.g. 1024 and 768).

To discover the dimensions of other monitors in a multi-monitor system, use SysGet.

To instead discover the width and height of the entire desktop (even if it spans multiple monitors), use the following example:

VirtualWidth := SysGet(78)

VirtualHeight := SysGet(79)

In addition, use SysGet to discover the work area of a monitor, which can be smaller than the monitor's total area because the taskbar and other registered desktop toolbars are excluded.

A_ScreenDPI Number of pixels per logical inch along the screen width. In a system with multiple display monitors, this value is the same for all monitors. On most systems this is 96; it depends on the system's text size (DPI) setting. See also the GUI's -DPIScale option.

Misc.

Variable Description

A_Clipboard Can be used to get or set the contents of the OS's clipboard. For details, see A_Clipboard.

A_Cursor

The type of mouse cursor currently being displayed. It will be one of the following words: AppStarting, Arrow, Cross, Help, IBeam, Icon, No, Size, SizeAll, SizeNESW, SizeNS, SizeNWSE, SizeWE, UpArrow, Wait, Unknown. The acronyms used with the size-type cursors are compass

directions, e.g. NESW = NorthEast+SouthWest. The hand-shaped cursors (pointing and grabbing) are classified as Unknown.

A_EventInfo

Contains additional information about the following events:

Mouse wheel hotkeys (WheelDown/Up/Left/Right)

OnMessage

Regular Expression Callouts

Note: Unlike variables such as A_ThisHotkey, each thread retains its own value for A_EventInfo. Therefore, if a thread is interrupted by another, upon being resumed it will still see its original/correct values in these variables.

A_EventInfo can also be set by the script, but can only accept unsigned integers within the range available to pointers (32-bit or 64-bit depending on the version of AutoHotkey).

A_LastError

This is usually the result from the OS's GetLastError() function after the script calls certain functions, including DllCall, Run/RunWait, File/Ini/Reg functions (where documented) and possibly others. A_LastError is a number between 0 and 4294967295 (always formatted as decimal, not hexadecimal). Zero (0) means success, but any other number means the call failed. Each number corresponds to a specific error condition (to get a list, search www.microsoft.com for "system error codes"). A_LastError is a per-thread setting; that is, interruptions by other threads cannot change it.

Assigning a value to A_LastError also causes the OS's SetLastError() function to be called.

True

False

Contain 1 and 0. They can be used to make a script more readable. For details, see Boolean Values.

These are actually keywords, not variables.

Loop

Variable Description

A_Index Can be used to get or set the number of the current loop iteration (a 64-bit integer). It contains 1 the first time the loop's body is executed. For the second time, it contains 2; and so on. If an inner loop is enclosed by an outer loop, the inner loop takes precedence. A_Index works inside all types of loops, but contains 0 outside of a loop. For counted loops such as Loop, changing A_Index affects the number of iterations that will be performed.

A_LoopFileName, etc. This and other related variables are valid only inside a file-loop.

A_LoopRegName, etc. This and other related variables are valid only inside a registry-loop.

A_LoopReadLine See file-reading loop.

A_LoopField See parsing loop.

Variable Capacity and Memory

When a variable is given a new string longer than its current contents, additional system memory is allocated automatically.

The memory occupied by a large variable can be freed by setting it equal to nothing, e.g. var := "".

There is no limit to how many variables a script may create. The program is designed to support at least several million variables without a significant drop in performance.

Functions and expressions that accept numeric inputs generally support 15 digits of precision for floating point values. For integers, 64-bit signed values are supported, which range from -9223372036854775808 (-0x8000000000000000) to 9223372036854775807 (0x7FFFFFFFFFFFFFFF). Any integer constants outside this range wrap around. Similarly, arithmetic operations on integers wrap around upon overflow (e.g. 0x7FFFFFFFFFFFFFFF + 1 = -0x8000000000000000).

Functions

Table of Contents

Introduction and Simple Examples

Parameters

Optional Parameters

Returning Values to Caller

Variadic Functions

Local Variables

Dynamically Calling a Function

Short-circuit Boolean Evaluation

Nested Functions

Return, Exit, and General Remarks

Using #Include to Share Functions Among Multiple Scripts

Built-in Functions

Introduction and Simple Examples

A function is a reusable block of code that can be executed by calling it. A function can optionally accept parameters (inputs) and return a value (output). Consider the following simple function that accepts two numbers and returns their sum:

```
Add(x, y)
{
   return x + y
}
```

The above is known as a function definition because it creates a function named "Add" (not case sensitive) and establishes that anyone who calls it must provide exactly two parameters (x and y). To call the function, assign its result to a variable with the := operator. For example:

```
Var := Add(2, 3)  ; The number 5 will be stored in Var.
```

Also, a function may be called without storing its return value:

```
Add(2, 3)
Add 2, 3  ; Parentheses can be omitted if used at the start of a line.
```

But in this case, any value returned by the function is discarded; so unless the function produces some effect other than its return value, the call would serve no purpose.

Within an expression, a function call "evaluates to" the return value of the function. The return value can be assigned to a variable as shown above, or it can be used directly as shown below:

```
if InStr(MyVar, "fox")
    MsgBox "The variable MyVar contains the word fox."
```

Parameters

When a function is defined, its parameters are listed in parentheses next to its name (there must be no spaces between its name and the open-parenthesis). If a function does not accept any parameters, leave the parentheses empty; for example: GetCurrentTimestamp().

ByRef Parameters: From the function's point of view, parameters are essentially the same as local variables unless they are marked as ByRef as in this example:

```
a := 1, b := 2
Swap(&a, &b)
MsgBox a ',' b

Swap(&Left, &Right)
{
    temp := Left
    Left := Right
    Right := temp
}
```

In the example above, the use of & requires the caller to pass a VarRef, which usually corresponds to one of the caller's variables. Each parameter becomes an alias for the variable represented by the VarRef. In other words, the parameter and the caller's variable both refer to the same contents in

memory. This allows the Swap function to alter the caller's variables by moving Left's contents into Right and vice versa.

By contrast, if ByRef were not used in the example above, Left and Right would be copies of the caller's variables and thus the Swap function would have no external effect. However, the function could instead be changed to explicitly dereference each VarRef. For example:

```
Swap(Left, Right)
{
   temp := %Left%
   %Left% := %Right%
   %Right% := temp
}
```

Since return can send back only one value to a function's caller, VarRefs can be used to send back extra results. This is achieved by having the caller pass in a reference to a variable (usually empty) in which the function stores a value.

When passing large strings to a function, ByRef enhances performance and conserves memory by avoiding the need to make a copy of the string. Similarly, using ByRef to send a long string back to the caller usually performs better than something like Return HugeString. However, what the function receives is not a reference to the string, but a reference to the variable. Future improvements might supersede the use of ByRef for these purposes.

Known limitations:

It is not possible to construct a VarRef for a property of an object (such as foo.bar), A_Clipboard or any other built-in variable, so those cannot be passed ByRef.

If a parameter in a function-call resolves to a variable (e.g. Var or ++Var or Var*=2), other parameters to its left or right can alter that variable before it is passed to the function. For example, MyFunc(Var, Var++) would unexpectedly pass 1 and 0 when Var is initially 0, because the first Var is not dereferenced until the function call is executed. Since this behavior is counterintuitive, it might change in a future release.

Optional Parameters

When defining a function, one or more of its parameters can be marked as optional.

Append := followed by a literal number, quoted/literal string such as "fox" or "", or an expression that should be evaluated each time the parameter needs to be initialized with its default value. For example, X:=[] would create a new Array each time.

Append ? or := unset to define a parameter which is unset by default.

The following function has its Z parameter marked optional:

```
Add(X, Y, Z := 0) {
   return X + Y + Z
}
```

When the caller passes three parameters to the function above, Z's default value is ignored. But when the caller passes only two parameters, Z automatically receives the value 0.

It is not possible to have optional parameters isolated in the middle of the parameter list. In other words, all parameters that lie to the right of the first optional parameter must also be marked optional. However, optional parameters may be omitted from the middle of the parameter list when calling the function, as shown below:

```
MyFunc(1,, 3)
MyFunc(X, Y:=2, Z:=0) {  ; Note that Z must still be optional in this case.
   MsgBox X ", " Y ", " Z
}
```

ByRef parameters also support default values; for example: MyFunc(&p1 := ""). Whenever the caller omits such a parameter, the function creates a local variable to contain the default value; in other words, the function behaves as though the symbol "&" is absent.

Unset Parameters

To mark a parameter as optional without supplying a default value, use the keyword unset or the ? suffix. In that case, whenever the parameter is omitted, the corresponding variable will have no value. Use IsSet to determine whether the parameter has been given a value, as shown below:

```
MyFunc(p?) {  ; Equivalent to MyFunc(p := unset)
   if IsSet(p)
      MsgBox "Caller passed " p
   else
      MsgBox "Caller did not pass anything"
}

MyFunc(42)
MyFunc
```

Attempting to read the parameter's value when it has none is considered an error, the same as for any uninitialized variable. To pass an optional parameter through to another function even when the parameter has no value, use the maybe operator (var?). For example:

```
Greet(title?) {
   MsgBox("Hello!", title?)
}

Greet "Greeting"  ; Title is "Greeting"
Greet             ; Title is A_ScriptName
```

Returning Values to Caller

As described in introduction, a function may optionally return a value to its caller.

```
MsgBox returnTest()


returnTest() {
   return 123
}
```

If you want to return extra results from a function, you may also use ByRef (&):

```
returnByRef(&A,&B,&C)
MsgBox A "," B "," C


returnByRef(&val1, &val2, val3)
{
   val1 := "A"
   val2 := 100
   %val3% := 1.1  ; % is used because & was omitted.
   return
}
```

Objects and Arrays can be used to return multiple values or even named values:

```
Test1 := returnArray1()
MsgBox Test1[1] "," Test1[2]


Test2 := returnArray2()
MsgBox Test2[1] "," Test2[2]


Test3 := returnObject()
```

```
MsgBox Test3.id "," Test3.val


returnArray1() {
   Test := [123,"ABC"]
   return Test
}


returnArray2() {
   x := 456
   y := "EFG"
   return [x, y]
}


returnObject() {
   Test := {id: 789, val: "HIJ"}
   return Test
}
```

Variadic Functions

When defining a function, write an asterisk after the final parameter to mark the function as variadic, allowing it to receive a variable number of parameters:

```
Join(sep, params*) {
   for index,param in params
      str .= param . sep
   return SubStr(str, 1, -StrLen(sep))
}
```

MsgBox Join("`n", "one", "two", "three")

When a variadic function is called, surplus parameters can be accessed via an object which is stored in the function's final parameter. The first surplus parameter is at params[1], the second at params[2] and so on. As it is an array, params.Length can be used to determine the number of parameters.

Attempting to call a non-variadic function with more parameters than it accepts is considered an error. To permit a function to accept any number of parameters without creating an array to store the surplus parameters, write * as the final parameter (without a parameter name).

Note: The "variadic" parameter can only appear at the end of the formal parameter list.

Variadic Function Calls

While variadic functions can accept a variable number of parameters, an array of parameters can be passed to any function by applying the same syntax to a function-call:

substrings := ["one", "two", "three"]

MsgBox Join("`n", substrings*)

Notes:

The object can be an Array or any other kind of enumerable object (any object with an __Enum method) or an enumerator. If the object is not an Array, __Enum is called with a count of 1 and the enumerator is called with only one parameter at a time.

Array elements with no value (such as the first element in [,2]) are equivalent to omitting the parameter; that is, the parameter's default value is used if it is optional, otherwise an exception is thrown.

This syntax can also be used when calling methods or setting or retrieving properties of objects; for example, Object.Property[Params*].

Known limitations:

Only the right-most parameter can be expanded this way. For example, MyFunc(x, y*) is supported but MyFunc(x*, y) is not.

There must not be any non-whitespace characters between the asterisk (*) and the symbol which ends the parameter list.

Function call statements cannot be variadic; that is, the parameter list must be enclosed in parentheses (or brackets for a property).

Local and Global Variables

Local Variables

Local variables are specific to a single function and are visible only inside that function. Consequently, a local variable may have the same name as a global variable but have separate contents. Separate functions may also safely use the same variable names.

All local variables which are not static are automatically freed (made empty) when the function returns, with the exception of variables which are bound to a closure or VarRef (such variables are freed at the same time as the closure or VarRef).

Built-in variables such as A_Clipboard and A_TimeIdle are never local (they can be accessed from anywhere), and cannot be redeclared. (This does not apply to built-in classes such as Object; they are predefined as global variables.)

Functions are assume-local by default. Variables accessed or created inside an assume-local function are local by default, with the following exceptions:

Global variables which are only read by the function, not assigned or used with the reference operator (&).

Nested functions may refer to local and static variables created by an enclosing function.

The default may also be overridden by declaring the variable using the local keyword, or by changing the mode of the function (as shown below).

Global variables

Any variable reference in an assume-local function may resolve to a global variable if it is only read. However, if a variable is used in an assignment or with the reference operator (&), it is automatically local by default. This allows functions to read global variables or call global or built-in functions without declaring them inside the function, while protecting the script from unintended side-effects when the name of a local variable being assigned coincides with a global variable. For example:

```
LogToFile(TextToLog)
{
   ; LogFileName was previously given a value somewhere outside this function.
   ; FileAppend is a predefined global variable containing a built-in function.
   FileAppend TextToLog "`n", LogFileName
}
```

Otherwise, to refer to an existing global variable inside a function (or create a new one), declare the variable as global prior to using it. For example:

```
SetDataDir(Dir)
{
   global LogFileName
   LogFileName := Dir . "\My.log"
   global DataDir := Dir  ; Declaration combined with assignment, described below.
}
```

Assume-global mode: If a function needs to access or create a large number of global variables, it can be defined to assume that all its variables are global (except its parameters) by making its first line the word "global". For example:

```
SetDefaults()
{
   global
```

```
    MyGlobal := 33  ; Assigns 33 to a global variable, first creating the variable if necessary.

    local x, y:=0, z  ; Local variables must be declared in this mode, otherwise they would be assumed global.

}
```

Static variables

Static variables are always implicitly local, but differ from locals because their values are remembered between calls. For example:

```
LogToFile(TextToLog)

{

    static LoggedLines := 0

    LoggedLines += 1  ; Maintain a tally locally (its value is remembered between calls).

    global LogFileName

    FileAppend LoggedLines ": " TextToLog "`n", LogFileName

}
```

A static variable may be initialized on the same line as its declaration by following it with := and any expression. For example: static X:=0, Y:="fox". Static declarations are evaluated the same as local declarations, except that after a static initializer (or group of combined initializers) is successfully evaluated, it is effectively removed from the flow of control and will not execute a second time.

Nested functions can be declared static to prevent them from capturing non-static local variables of the outer function.

Assume-static mode: A function may be defined to assume that all its undeclared local variables are static (except its parameters) by making its first line the word "static". For example:

```
GetFromStaticArray(WhichItemNumber)

{
```

```
    static
    static FirstCallToUs := true  ; Each static declaration's initializer still runs only once.
    if FirstCallToUs  ; Create a static array during the first call, but not on subsequent calls.
    {
        FirstCallToUs := false
        StaticArray := []
        Loop 10
            StaticArray.Push("Value #" . A_Index)
    }
    return StaticArray[WhichItemNumber]
}
```

In assume-static mode, any variable that should not be static must be declared as local or global (with the same exceptions as for assume-local mode).

More about locals and globals

Multiple variables may be declared on the same line by separating them with commas as in these examples:

```
global LogFileName, MaxRetries := 5
static TotalAttempts := 0, PrevResult
```

A variable may be initialized on the same line as its declaration by following it with an assignment. Unlike static initializers, the initializers of locals and globals execute every time the function is called. In other words, a line like local x := 0 has the same effect as writing two separate lines: local x followed by x := 0. Any assignment operator can be used, but a compound assignment such as global HitCount += 1 would require that the variable has previously been assigned a value.

Because the words local, global, and static are processed immediately when the script launches, a variable cannot be conditionally declared by means of an IF statement. In other words, a declaration inside an IF's or ELSE's block takes effect unconditionally for the entire function (but any initializers included in the declaration are still conditional). A dynamic declaration such as

global Array%i% is not possible, since all non-dynamic references to variables such as Array1 or Array99 would have already been resolved to addresses.

Dynamically Calling a Function

Although a function call expression usually begins with a literal function name, the target of the call can be any expression which produces a function object. In the expression GetKeyState("Shift"), GetKeyState is actually a variable reference, although it usually refers to a read-only variable containing a built-in function.

A function call is said to be dynamic if the target of the call is determined while the script is running, instead of before the script starts. The same syntax is used as for normal function calls; the only apparent difference is that certain error-checking is performed at load time for non-dynamic calls but only at run time for dynamic calls.

For example, MyFunc() would call the function object contained by MyFunc, which could be either the actual name of a function, or just a variable which has been assigned a function.

Other expressions can be used as the target of a function call, including double-derefs. For example, MyArray[1]() would call the function contained by the first element of MyArray, while %MyVar%() would call the function contained by the variable whose name is contained by MyVar. In other words, the expression preceding the parameter list is first evaluated to get a function object, then that object is called.

If the target value cannot be called due to one of the reasons below, an Error is thrown:

If the target value is not of a type that can be called, a MethodError is thrown. Any value with a Call method can be called, so HasMethod(value, "Call") can be used to avoid this error.

Passing too few or too many parameters, which can often be avoided by checking the function's MinParams, MaxParams and IsVariadic properties.

Passing something other than a variable reference (VarRef) to a ByRef or OutputVar parameter, which could be avoided through the use of the IsByRef method.

The caller of a function should generally know what each parameter means and how many there are before calling the function. However, for dynamic calls, the function is often written to suit the

function call, and in such cases failure might be caused by a mistake in the function definition rather than incorrect parameter values.

Short-circuit Boolean Evaluation

When AND, OR, and the ternary operator are used within an expression, they short-circuit to enhance performance (regardless of whether any function calls are present). Short-circuiting operates by refusing to evaluate parts of an expression that cannot possibly affect its final result. To illustrate the concept, consider this example:

```
if (ColorName != "" AND not FindColor(ColorName))
   MsgBox ColorName " could not be found."
```

In the example above, the FindColor() function never gets called if the ColorName variable is empty. This is because the left side of the AND would be false, and thus its right side would be incapable of making the final outcome true.

Because of this behavior, it's important to realize that any side-effects produced by a function (such as altering a global variable's contents) might never occur if that function is called on the right side of an AND or OR.

It should also be noted that short-circuit evaluation cascades into nested ANDs and ORs. For example, in the following expression, only the leftmost comparison occurs whenever ColorName is blank. This is because the left side would then be enough to determine the final answer with certainty:

```
if (ColorName = "" OR FindColor(ColorName, Region1) OR FindColor(ColorName, Region2))
   break   ; Nothing to search for, or a match was found.
```

As shown by the examples above, any expensive (time-consuming) functions should generally be called on the right side of an AND or OR to enhance performance. This technique can also be used to prevent a function from being called when one of its parameters would be passed a value it considers inappropriate, such as an empty string.

The ternary conditional operator (?:) also short-circuits by not evaluating the losing branch.

Nested Functions

A nested function is one defined inside another function. For example:

```
outer(x) {
   inner(y) {
      MsgBox(y, x)
   }
   inner("one")
   inner("two")
}
outer("title")
```

A nested function is not accessible by name outside of the function which immediately encloses it, but is accessible anywhere inside that function, including inside other nested functions (with exceptions).

By default, a nested function may access any static variable of any function which encloses it, even dynamically. However, a non-dynamic assignment inside a nested function typically resolves to a local variable if the outer function has neither a declaration nor a non-dynamic assignment for that variable.

By default, a nested function automatically "captures" a non-static local variable of an outer function when the following requirements are met:

The outer function must refer to the variable in at least one of the following ways:

By declaring it with local, or as a parameter or nested function.

As the non-dynamic target of an assignment or the reference operator (&).

The inner function (or a function nested inside it) must refer to the variable non-dynamically.

A nested function which has captured variables is known as a closure.

Non-static local variables of the outer function cannot be accessed dynamically unless they have been captured.

Explicit declarations always take precedence over local variables of the function which encloses them. For example, local x declares a variable local to the current function, independent of any x in the outer function. Global declarations in the outer function also affect nested functions, except where overridden by an explicit declaration.

If a function is declared assume-global, any local or static variables created outside that function are not directly accessible to the function itself or any of its nested functions. By contrast, a nested function which is assume-static can still refer to variables from the outer function, unless the function itself is declared static.

Functions are assume-local by default, and this is true even for nested functions, even those inside an assume-static function. However, if the outer function is assume-global, nested functions behave as though assume-global by default, except that they can refer to local and static variables of the outer function.

Each function definition creates a read-only variable containing the function itself; that is, a Func or Closure object. See below for examples of how this might be used.

Static functions

Any nested function which does not capture variables is automatically static; that is, every call to the outer function references the same Func. The keyword static can be used to explicitly declare a nested function as static, in which case any non-static local variables of the outer function are ignored. For example:

```
outer() {
   x := "outer value"
   static inner() {
```

```
    x := "inner value"  ; Creates a variable local to inner
    MsgBox type(inner)  ; Displays "Func"
  }
  inner()
  MsgBox x  ; Displays "outer value"
}
outer()
```

A static function cannot refer to other nested functions outside its own body unless they are explicitly declared static. Note that even if the function is assume-static, a non-static nested function may become a closure if it references a function parameter.

Closures

A closure is a nested function bound to a set of free variables. Free variables are local variables of the outer function which are also used by nested functions. Closures allow one or more nested functions to share variables with the outer function even after the outer function returns.

To create a closure, simply define a nested function which refers to variables of the outer function. For example:

```
make_greeter(f)
{
  greet(subject)  ; This will be a closure due to f.
  {
    MsgBox Format(f, subject)
  }
  return greet  ; Return the closure.
}
```

```
g := make_greeter("Hello, {}!")
g(A_UserName)
g("World")
```

Closures may also be used with built-in functions, such as SetTimer or Hotkey. For example:

```
app_hotkey(keyname, app_title, app_path)
{
   activate(keyname)  ; This will be a closure due to app_title and app_path.
   {
      if WinExist(app_title)
         WinActivate
      else
         Run app_path
   }
   Hotkey keyname, activate
}
; Win+N activates or launches Notepad.
app_hotkey "#n", "ahk_class Notepad", "notepad.exe"
; Win+W activates or launches WordPad.
app_hotkey "#w", "ahk_class WordPadClass", "wordpad.exe"
```

A nested function is automatically a closure if it captures any non-static local variables of the outer function. The variable corresponding to the closure itself (such as activate) is also a non-static local variable, so any nested functions which refer to a closure are automatically closures.

Each call to the outer function creates new closures, distinct from any previous calls.

It is best not to store a reference to a closure in any of the closure's own free variables, since that creates a circular reference which must be broken (such as by clearing the variable) before the closure can be freed. However, a closure may safely refer to itself and other closures by their original variables without creating a circular reference. For example:

```
timertest() {
   x := "tock!"
   tick() {
      MsgBox x          ; x causes this to become a closure.
      SetTimer tick, 0   ; Using the closure's original var is safe.
      ; SetTimer t, 0    ; Capturing t would create a circular reference.
   }
   t := tick            ; This is okay because t isn't captured above.
   SetTimer t, 1000
}
timertest()
```

Return, Exit, and General Remarks

If the flow of execution within a function reaches the function's closing brace prior to encountering a Return, the function ends and returns a blank value (empty string) to its caller. A blank value is also returned whenever the function explicitly omits Return's parameter.

When a function uses Exit to terminate the current thread, its caller does not receive a return value at all. For example, the statement Var := Add(2, 3) would leave Var unchanged if Add() exits. The same thing happens if the function is exited because of Throw or a runtime error (such as running a nonexistent file).

To call a function with one or more blank values (empty strings), use an empty pair of quotes as in this example: FindColor(ColorName, "").

Since calling a function does not start a new thread, any changes made by a function to settings such as SendMode and SetTitleMatchMode will go into effect for its caller too.

When used inside a function, ListVars displays a function's local variables along with their contents. This can help debug a script.

Style and Naming Conventions

You might find that complex functions are more readable and maintainable if their special variables are given a distinct prefix. For example, naming each parameter in a function's parameter list with a leading "p" or "p_" makes their special nature easy to discern at a glance, especially when a function has several dozen local variables competing for your attention. Similarly, the prefix "r" or "r_" could be used for ByRef parameters, and "s" or "s_" could be used for static variables.

The One True Brace (OTB) style may optionally be used to define functions. For example:

```
Add(x, y) {
   return x + y
}
```

Using #Include to Share Functions Among Multiple Scripts

The #Include directive may be used to load functions from an external file.

Built-in Functions

A built-in function is overridden if the script defines its own function of the same name. For example, a script could have its own custom WinExist function that is called instead of the standard one. However, the script would then have no way to call the original function.

External functions that reside in DLL files may be called with DllCall.

To get a list of all built-in functions, see Alphabetical Function Index.

Labels

Table of Contents

Syntax and Usage

Look-alikes

Dynamic Labels

Named Loops

Related

Syntax and Usage

A label identifies a line of code, and can be used as a Goto target or to specify a loop to break out of or continue. A label consist of a name followed by a colon:

this_is_a_label:

Aside from whitespace and comments, no other code can be written on the same line as a label.

Names: Label names are not case sensitive (for ASCII letters), and may consist of letters, numbers, underscore and non-ASCII characters. For example: MyListView, Menu_File_Open, and outer_loop.

Scope: Each function has its own list of local labels. Inside a function, only that function's labels are visible/accessible to the script.

Target: The target of a label is the next line of executable code. Executable code includes functions, assignments, expressions and blocks, but not directives, labels, hotkeys or hotstrings. In the following example, run_notepad_1 and run_notepad_2 both point at the Run line:

run_notepad_1:

run_notepad_2:

```
    Run "notepad"
    return
```

Execution: Like directives, labels have no effect when reached during normal execution.

Look-alikes

Hotkey and hotstring definitions look similar to labels, but are not labels.

Hotkeys consist of a hotkey followed by double-colon.

^a::

Hotstrings consist of a colon, zero or more options, another colon, an abbreviation and double-colon.

:*:btw::

Dynamic Labels

In some cases a variable can be used in place of a label name. In such cases, the name stored in the variable is used to locate the target label. However, performance is slightly reduced because the target label must be "looked up" each time rather than only once when the script is first loaded.

Named Loops

A label can also be used to identify a loop for the Continue and Break statements. This allows the script to easily continue or break out of any number of nested loops.

Related

Functions, IsLabel, Goto, Break, Continue

Threads

The current thread is defined as the flow of execution invoked by the most recent event; examples include hotkeys, SetTimer subroutines, custom menu items, and GUI events. The current thread can be executing functions within its own subroutine or within other subroutines called by that subroutine.

Although AutoHotkey doesn't actually use multiple threads, it simulates some of that behavior: If a second thread is started -- such as by pressing another hotkey while the previous is still running -- the current thread will be interrupted (temporarily halted) to allow the new thread to become current. If a third thread is started while the second is still running, both the second and first will be in a dormant state, and so on.

When the current thread finishes, the one most recently interrupted will be resumed, and so on, until all the threads finally finish. When resumed, a thread's settings for things such as SendMode and SetKeyDelay are automatically restored to what they were just prior to its interruption; in other words, a thread will experience no side-effects from having been interrupted (except for a possible change in the active window).

Note: The KeyHistory function/menu-item shows how many threads are in an interrupted state and the ListHotkeys function/menu-item shows which hotkeys have threads.

A single script can have multiple simultaneous MsgBox, InputBox, FileSelect, and DirSelect dialogs. This is achieved by launching a new thread (via hotkey, timed subroutine, custom menu item, etc.) while a prior thread already has a dialog displayed.

By default, a given hotkey or hotstring subroutine cannot be run a second time if it is already running. Use #MaxThreadsPerHotkey to change this behavior.

Related: The Thread function sets the priority or interruptibility of threads.

Thread Priority

Any thread (hotkey, timed subroutine, custom menu item, etc.) with a priority lower than that of the current thread cannot interrupt it. During that time, such timers will not run, and any attempt by the user to create a thread (such as by pressing a hotkey or GUI button) will have no effect, nor

will it be buffered. Because of this, it is usually best to design high priority threads to finish quickly, or use Critical instead of making them high priority.

The default priority is 0. All threads use the default priority unless changed by one of the following methods:

A timed subroutine is given a specific priority via SetTimer.

A hotkey is given a specific priority via the Hotkey function.

A hotstring is given a specific priority when it is defined, or via the #Hotstring directive.

A custom menu item is given a specific priority via the Menu.Add method.

The current thread sets its own priority via the Thread function.

The OnExit callback function (if any) will always run when called for, regardless of the current thread's priority.

Thread Interruptibility

For most types of events, new threads are permitted to launch only if the current thread is interruptible. A thread can be uninterruptible for a number of reasons, including:

The thread has been marked as critical. Critical may have been called by the thread itself or by the auto-execute thread.

The thread has not been running long enough to meet the conditions for becoming interruptible, as set by Thread "Interrupt".

One of the script's menus is being displayed (such as the tray icon menu or a menu bar).

A delay is being performed by Send (most often due to SetKeyDelay), WinActivate, or a Clipboard operation.

An OnExit thread is executing.

A warning dialog is being displayed due to the A_MaxHotkeysPerInterval limit being reached, or due to a problem activating the keyboard or mouse hook (very rare).

Behavior of Uninterruptible Threads

Unlike high-priority threads, events that occur while the thread is uninterruptible are not discarded. For example, if the user presses a hotkey while the current thread is uninterruptible, the hotkey is buffered indefinitely until the current thread finishes or becomes interruptible, at which time the hotkey is launched as a new thread.

Any thread may be interrupted in emergencies. Emergencies consist of:

An OnExit callback.

Any OnMessage function that monitors a message which is not buffered.

Any callback indirectly triggered by the thread itself (e.g. via SendMessage or DllCall).

To avoid these interruptions, temporarily disable such functions.

A critical thread becomes interruptible when a MsgBox or other dialog is displayed. However, unlike Thread Interrupt, the thread becomes critical (and therefore uninterruptible) again after the user dismisses the dialog.

Changes from v1.1 to v2.0

Language

Legacy Syntax Removed

Removed literal assignments: var = value

Removed all legacy If statements, leaving only if expression, which never requires parentheses (but allows them, as in any expression).

Removed "command syntax". There are no "commands", only function call statements, which are just function or method calls without parentheses. That means:

All former commands are now functions (excluding control flow statements).

All functions can be called without parentheses if the return value is not needed (but as before, parentheses cannot be omitted for calls within an expression).

All parameters are expressions, so all text is "quoted" and commas never need to be escaped. Currently this excludes a few directives (which are neither commands nor functions).

Parameters are the same regardless of parentheses; i.e. there is no output variable for the return value, so it is discarded if parentheses are omitted.

Normal variable references are never enclosed in percent signs (except with #Include and #DllLoad). Use concatenation or Format to include variables in text.

There is no comma between the function name and parameters, so MouseGetPos(, y) = MouseGetPos , y (x is omitted). A space or tab is required for clarity. For consistency, directives also follow the new convention (there must not be a comma between the directive name and parameter).

There is no percent-space prefix to force an expression. Unquoted percent signs in expressions are used only for double-derefs/dynamic references, and having an odd number of them is a syntax error.

Method call statements (method calls which omit parentheses) are restricted to a plain variable followed by one or more identifiers separated by dots, such as MyVar.MyProperty.MyMethod "String to pass".

The translation from v1-command to function is generally as follows (but some functions have been changed, as documented further below):

If the command's first parameter is an output variable and the second parameter is not, it becomes the return value and is removed from the parameter list.

The remaining output variables are handled like ByRef parameters (for which usage and syntax has changed), except that they permit references to writable built-in variables.

An exception is thrown on failure instead of setting ErrorLevel.

Values formerly returned via ErrorLevel are returned by other means, replaced with exceptions, superseded or simply not returned.

All control flow statements also accept expressions, except where noted below.

All control flow statements which take parameters (currently excluding the two-word Loop statements) support parentheses around their parameter list, without any space between the name

and parenthesis. For example, return(var). However, these are not functions; for instance, x := return(y) is not valid. If and While already supported this.

Loop (except Loop Count) is now followed by a secondary keyword (Files, Parse, Read or Reg) which cannot be "quoted" or contained by a variable. Currently the keyword can be followed by a comma, but it is not required as this is not a parameter. OTB is supported by all modes.

Goto, break and continue require an unquoted label name, similar to v1 (goto label jumps to label:). To jump to a label dynamically, use parentheses immediately after the name: goto(expression). However, this is not a function and cannot be used mid-expression. Parentheses can be used with break or continue, but in that case the parameter must be a single literal number or quoted string.

Gosub has been removed, and labels can no longer be used with functions such as SetTimer and Hotkey.

They were redundant; basically just a limited form of function, without local variables or a return value, and being in their own separate namespace. Functions can be used everywhere that label subroutines were used before (even inside other functions).

Functions cannot overlap (but can be contained within a function). Instead, use multiple functions and call one from the other. Instead of A_ThisLabel, use function parameters.

Unlike subroutines, if one forgets to define the end of a function, one is usually alerted to the error as each { must have a corresponding }. It may also be easier to identify the bounds of a function than a label subroutine.

Functions can be placed in the auto-execute section without interrupting it. The auto-execute section can now easily span the entire script, so may instead be referred to as global code, executing within the auto-execute thread.

Functions might be a little less prone to being misused as "goto" (where a user gosubs the current subroutine in order to loop, inevitably exhausting stack space and terminating the program).

There is less ambiguity without functions (like Hotkey) accepting a label or a function, where both can exist with the same name at once.

For all remaining uses of labels, it is not valid to refer to a global label from inside a function. Therefore, label lookup can be limited to the local label list. Therefore, there is no need to check for invalid jumps from inside a function to outside (which were never supported).

Hotkey and Hotstring Labels

Hotkeys and non-autoreplace hotstrings are no longer labels; instead, they (automatically) define a function. For multi-line hotkeys, use braces to enclose the body of the hotkey instead of terminating it with return (which is implied by the ending brace). To allow a hotkey to be called explicitly, specify funcName(ThisHotkey) between the :: and { - this can also be done in v1.1.20+, but now there is a parameter. When the function definition is not explicit, the parameter is named ThisHotkey.

Note: Hotkey functions are assume-local by default and therefore cannot assign to global variables without a declaration.

Names

Function and variable names are now placed in a shared namespace.

Each function definition creates a constant (read-only variable) within the current scope.

Use MyFunc in place of Func("MyFunc").

Use MyFunc in place of "MyFunc" when passing the function to any built-in function such as SetTimer or Hotkey. Passing a name (string) is no longer supported.

Use myVar() in place of %myVar%() when calling a function by value.

To call a function when all you have is a function name (string), first use a double-deref to resolve the name to a variable and retrieve its value (the function object). %myVar%() now actually performs a double-deref and then calls the result, equivalent to f := %myVar%, f(). Avoid handling functions by name (string) where possible; use references instead.

Names cannot start with a digit and cannot contain the following characters which were previously allowed: @ # $. Only letters, numbers, underscore and non-ASCII characters are allowed.

Reserved words: Declaration keywords and names of control flow statements cannot be used as variable, function or class names. This includes local, global, static, if, else, loop, for, while, until, break, continue, goto, return, switch, case, try, catch, finally and throw. The purpose of this is primarily to detect errors such as if (ex) break.

Reserved words: as, and, contains, false, in, is, IsSet, not, or, super, true, unset. These words are reserved for future use or other specific purposes, and are not permitted as variable or function names even when unambiguous. This is primarily for consistency: in v1, and := 1 was allowed on its own line but (and := 1) would not work.

The words listed above are permitted as property or window group names. Property names in typical use are preceded by ., which prevents the word from being interpreted as an operator. By contrast, keywords are never interpreted as variable or function names within an expression. For example, not(x) is equivalent to not (x) or (not x).

A number of classes are predefined, effectively reserving those global variable names in the same way that a user-defined class would. (However, the changes to scope described below mitigate most issues arising from this.) For a list of classes, see Built-in Classes.

Scope

Super-global variables have been removed (excluding built-in variables, which aren't quite the same as they cannot be redeclared or shadowed).

Within an assume-local function, if a given name is not used in a declaration or as the target of a non-dynamic assignment or the reference (&) operator, it may resolve to an existing global variable.

In other words:

Functions can now read global variables without declaring them.

Functions which have no global declarations cannot directly modify global variables (eliminating one source of unintended side-effects).

Adding a new class to the script is much less likely to affect the behaviour of any existing function, as classes are not super-global.

The global keyword is currently redundant when used in global scope, but can be used for clarity. Variables declared this way are now much less likely to conflict with local variables (such as when combining scripts manually or with #Include), as they are not super-global. On the other hand, some convenience is lost.

Declarations are generally not needed as much.

Force-local mode has been removed.

Variables

Local static variables are initialized if and when execution reaches them, instead of being executed in linear order before the auto-execute section begins. Each initializer has no effect the second time it is reached. Multiple declarations are permitted and may execute for the same variable at different times. There are multiple benefits:

When a static initializer calls other functions with static variables, there is less risk of initializers having not executed yet due to the order of the function definitions.

Because the function has been called, parameters, A_ThisFunc and closures are available (they previously were not).

A static variable can be initialized conditionally, adding flexibility, while still only executing once without requiring if IsSet().

Since there may be multiple initializers for a single static variable, compound assignments such as static x += 1 are permitted. (This change reduced code size marginally as it was already permitted by local and global.)

Note: static init := somefunction() can no longer be used to auto-execute somefunction. However, since label-and-return based subroutines can now be completely avoided, the auto-execute section is able to span the entire script.

Declaring a variable with local no longer makes the function assume-global.

Double-derefs are now more consistent with variables resolved at load-time, and are no longer capable of creating new variables. This avoids some inconsistencies and common points of confusion.

Double-derefs which fail for any reason now cause an error to be thrown. Previously any cases with an invalid name would silently produce an empty string, while other cases would create and return an empty variable.

Expressions

Quoted literal strings can be written with "double" or 'single' quote marks, but must begin and end with the same mark. Literal quote marks are written by preceding the mark with an escape character - `" or `' - or by using the opposite type of quote mark: '"42" is the answer'. Doubling the quote marks has no special meaning, and causes an error since auto-concat requires a space.

The operators &&, ||, and and or yield whichever value determined the result, similar to JavaScript and Lua. For example, "" or "default" yields "default" instead of 1. Scripts which require a pure boolean value (0 or 1) can use something like !!(x or y) or (x or y) ? 1 : 0.

Auto-concat now requires at least one space or tab in all cases (the v1 documentation says there "should be" a space).

The result of a multi-statement expression such as x(), y() is the last (right-most) sub-expression instead of the first (left-most) sub-expression. In both v1 and v2, the sub-expressions are evaluated in left to right order.

Equals after a comma is no longer assignment: y=z in x:=y, y=z is an ineffectual comparison instead of an assignment.

:= += -= *= /= ++ -- have consistent behaviour regardless of whether they are used on their own or combined with other operators, such as with x := y, y += 2. Previously, there were differences in behaviour when an error occurred within the expression or a blank value was used in a math operation.

!= is now always case-insensitive, like =, while !== has been added as the counterpart of ==.

<> has been removed.

// now throws an exception if given a floating-point number. Previously the results were inconsistent between negative floats and negative integers.

|, ^, &, << and >> now throw an exception if given a floating-point number, instead of truncating to integer.

Scientific notation can be used without a decimal point (but produces a floating-point number anyway). Scientific notation is also supported when numeric strings are converted to integers (for example, "1e3" is interpreted as 1000 instead of 1).

Function calls now permit virtually any sub-expression for specifying which function to call, provided that there is no space or tab before the open-parenthesis of the parameter list. For example, MyFunc() would call the value MyFunc regardless of whether that is the function's actual name or a variable containing a function object, and (a?b:c)() would call either b or c depending on a. Note that x.y() is still a method call roughly equivalent to (x.y)(x), but a[i]() is now equivalent to (a[i])().

Double-derefs now permit almost any expression (not just variables) as the source of the variable name. For example, DoNotUseArray%n+1% and %(%triple%)% are valid. Double-deref syntax is now also used to dereference VarRefs, such as ref := &var, value := %ref%.

The expressions funcName[""]() and funcName.() no longer call a function by name. Omitting the method name as in .() now causes a load-time error message. Functions should be called or handled by reference, not by name.

var := with no r-value is treated as an error at load-time. In v1 it was equivalent to var := "", but silently failed if combined with another expression - for example: x :=, y :=.

Where a literal string is followed by an ambiguous unary/binary operator, an error is reported at load-time. For instance, "new counter:" ++counter is probably supposed to increment and display counter, but technically it is invalid addition and unary plus.

word ++ and word -- are no longer expressions, since word can be a user-defined function (and ++/- may be followed by an expression which produces a variable reference). To write a

standalone post-increment or post-decrement expression, either omit the space between the variable and the operator, or wrap the variable or expression in parentheses.

word ? x : y is still a ternary expression, but more complex cases starting with a word, such as word1 word2 ? x : y, are always interpreted as function calls to word1 (even if no such function exists). To write a standalone ternary expression with a complex condition, enclose the condition in parentheses.

The new is operator such as in x is y can be used to check whether value x is an instance of class y, where y must be an Object with a prototype property (i.e. a Class). This includes primitive values, as in x is Integer (which is strictly a type check, whereas IsInteger(x) checks for potential conversion).

Keywords contains and in are reserved for future use.

&var (address-of) has been replaced with StrPtr(var) and ObjPtr(obj) to more clearly show the intent and enhance error checking. In v1, address-of returned the address of var's internal string buffer, even if it contained a number (but not an object). It was also used to retrieve the address of an object, and getting an address of the wrong type can have dire consequences.

&var is now the reference operator, which is used with all ByRef and OutputVar parameters to improve clarity and flexibility (and make other language changes possible). See Variable References (VarRef) for more details.

String length is now cached during expression evaluation. This improves performance and allows strings to contain binary zero. In particular:

Concatenation of two strings where one or both contain binary zero no longer causes truncation of the data.

The case-sensitive equality operators (== and !==) can be used to compare binary data. The other comparison operators only "see" up to the first binary zero.

Binary data can be returned from functions and assigned to objects.

Most functions still expect null-terminated strings, so will only "see" up to the first binary zero. For example, MsgBox would display only the portion of the string before the first binary zero.

The * (deref) operator has been removed. Use NumGet instead.

The ~ (bitwise-NOT) operator now always treats its input as a 64-bit signed integer; it no longer treats values between 0 and 4294967295 as unsigned 32-bit.

>>> and >>>= have been added for logical right bit shift.

Added fat arrow functions. The expression Fn(Parameters) => Expression defines a function named Fn (which can be blank) and returns a Func or Closure object. When called, the function evaluates Expression and returns the result. When used inside another function, Expression can refer to the outer function's variables (this can also be done with a normal function definition).

The fat arrow syntax can also be used to define methods and property getters/setters (in which case the method/property definition itself isn't an expression, but its body just returns an expression).

Literal numbers are now fully supported on the left-hand side of member access (dot). For example, 0.1 is a number but 0.min and 0.1.min access the min property which can be handled by a base object (see Primitive Values). 1..2 or 1.0.2 is the number 1.0 followed by the property 2. Example use might be to implement units of measurement, literal version numbers or ranges.

x**y: Where x and y are integers and y is positive, the power operator now gives correct results for all inputs if in range, where previously some precision was lost due to the internal use of floating-point math. Behaviour of overflow is undefined.

Objects (Misc)

See also: Objects

There is now a distinction between properties accessed with . and data (items, array or map elements) accessed with []. For example, dictionary["Count"] can return the definition of "Count" while dictionary.Count returns the number of words contained within. User-defined objects can utilize this by defining an __Item property.

When the name of a property or method is not known in advance, it can (and must) be accessed by using percent signs. For example, obj.%varname%() is the v2 equivalent of obj[varname](). The use of [] is reserved for data (such as array elements).

The literal syntax for constructing an ad hoc object is still basically {name: value}, but since plain objects now only have "properties" and not "array elements", the rules have changed slightly for consistency with how properties are accessed in other contexts:

o := {a: b} uses the name "a", as before.

o := {%a%: b} uses the property name in a, instead of taking that as a variable name, performing a double-deref, and using the contents of the resulting variable. In other words, it has the same effect as o := {}, o.%a% := b.

Any other kind of expression to the left of : is illegal. For instance, {(a): b} or {an error: 1}.

The use of the word "base" in base.Method() has been replaced with super (super.Method()) to distinguish the two concepts better:

super. or super[calls the super-class version of a method/property, where "super-class" is the base of the prototype object which was originally associated with the current function's definition.

super is a reserved word; attempting to use it without the . or [or (suffix or outside of a class results in a load time error.

base is a pre-defined property which gets or sets the object's immediate base object (like ObjGetBase/ObjSetBase). It is just a normal property name, not reserved.

Invoking super.x when the superclass has no definition of x throws an error, whereas base.x was previously ignored (even if it was an assignment).

Where Fn is an object, Fn() (previously written as %Fn%()) now calls Fn.Call() instead of Fn.() (which can now only be written as Fn.%""%()). Functions no longer support the nameless method.

this.Method() calls Fn.Call(this) (where Fn is the function object which implements the method) instead of Fn[this]() (which in v1, would result in a call to Fn.__Call(this) unless Fn[this] contains a function). Function objects should implement a Call method instead of __Call, which is only for explicit method calls.

Classname() (formerly new Classname()) now fails to create the object if the __New method is defined and it could not be called (e.g. because the parameter count is incorrect), or if parameters were passed and __New is not defined.

Objects created within an expression or returned from a function are now held until expression evaluation is complete, and then released. This improves performance slightly and allows temporary objects to be used for memory management within an expression, without fear of the objects being freed prematurely.

Objects can contain string values (but not keys) which contain binary zero. Cloning an object preserves binary data in strings, up to the stored length of the string (not its capacity). Historically, data was written beyond the value's length when dealing with binary data or structs; now, a Buffer object should be used instead.

Assignment expressions such as x.y := z now always yield the value of z, regardless of how x.y is implemented. The return value of a property setter is now ignored. Previously:

Some built-in objects returned z, some returned x.y (such as c := GuiObj.BackColor := "red" setting c to 0xFF0000), and some returned an incorrect value.

User-defined property setters may have returned unexpected values or failed to return anything.

x.y(z) := v is now a syntax error. It was previously equivalent to x.y[z] := v. In general, x.y(z) (method call) and x.y[z] (parameterized property) are two different operations, although they may be equivalent if x is a COM object (due to limitations of the COM interface).

Concatenating an object with another value or passing it to Loop is currently treated as an error, whereas previously the object was treated as an empty string. This may be changed to implicitly call .ToString(). Use String(x) to convert a value to a string; this calls .ToString() if x is an object.

When an object is called via IDispatch (the COM interface), any uncaught exceptions which cannot be passed back to the caller will cause an error dialog. (The caller may or may not show an additional error dialog without any specific details.) This also applies to event handlers being called due to the use of ComObjConnect.

Functions

Functions can no longer be dynamically called with more parameters than they formally accept.

Variadic functions are not affected by the above restriction, but normally will create an array each time they are called to hold the surplus parameters. If this array is not needed, the parameter name can now be omitted to prevent it from being created:

```
AcceptsOneOrMoreArgs(first, *) {
  ...
}
```

This can be used for callbacks where the additional parameters are not needed.

Variadic function calls now permit any enumerable object, where previously they required a standard Object with sequential numeric keys. If the enumerator returns more than one value per iteration, only the first value is used. For example, Array(mymap*) creates an array containing the keys of mymap.

Variadic function calls previously had half-baked support for named parameters. This has been disabled, to remove a possible impediment to the proper implementation of named parameters.

User-defined functions may use the new keyword unset as a parameter default value to make the parameter "unset" when no value was provided. The function can then use IsSet() to determine if a value was provided. unset is currently not permitted in any other context.

Scripts are no longer automatically included from the function library (Lib) folders when a function call is present without a definition, due to increased complexity and potential for accidents (now

that the MyFunc in MyFunc() can be any variable). #Include <LibName> works as before. It may be superseded by module support in a future release.

Variadic built-in functions now have a MaxParams value equal to MinParams, rather than an arbitrary number (such as 255 or 10000). Use IsVariadic to detect when there is no upper bound.

ByRef

ByRef parameters are now declared using &param instead of ByRef param, with some differences in usage.

ByRef parameters no longer implicitly take a reference to the caller's variable. Instead, the caller must explicitly pass a reference with the reference operator (&var). This allows more flexibility, such as storing references elsewhere, accepting them with a variadic function and passing them on with a variadic call.

When a parameter is marked ByRef, any attempt to explicitly pass a non-VarRef value causes an error to be thrown. Otherwise, the function can check for a reference with param is VarRef, check if the target variable has a value with IsSetRef(param), and explicitly dereference it with %param%.

ByRef parameters are now able to receive a reference to a local variable from a previous instance of the same function, when it is called recursively.

Nested Functions

One function may be defined inside another. A nested function may automatically "capture" non-static local variables from the enclosing function (under the right conditions), allowing them to be used after the enclosing function returns.

The new "fat arrow" => operator can also be used to create nested functions.

For full detail, see Nested Functions.

Uncategorized

:= must be used in place of = when initializing a declared variable or optional parameter.

return %var% now does a double-deref; previously it was equivalent to return var.

#Include is relative to the directory containing the current file by default. Its parameter may now optionally be enclosed in quote marks.

#ErrorStdOut's parameter may now optionally be enclosed in quote marks.

Label names are now required to consist only of letters, numbers, underscore and non-ASCII characters (the same as variables, functions, etc.).

Labels defined in a function have local scope; they are visible only inside that function and do not conflict with labels defined elsewhere. It is not possible for local labels to be called externally (even by built-in functions). Nested functions can be used instead, allowing full use of local variables.

for k, v in obj:

How the object is invoked has changed. See Enumeration.

for now restores k and v to the values they had before the loop began, after the loop breaks or completes.

An exception is thrown if obj is not an object or there is a problem retrieving or calling its enumerator.

Up to 19 variables can be used.

Variables can be omitted.

Escaping a comma no longer has any meaning. Previously if used in an expression within a command's parameter and not within parentheses, it forced the comma to be interpreted as the multi-statement operator rather than as a delimiter between parameters. It only worked this way for commands, not functions or variable declarations.

The escape sequence `s is now allowed wherever `t is supported. It was previously only allowed by #IfWin and (Join.

/ can now be placed at the end of a line to end a multi-line comment, to resolve a common point of confusion relating to how / */ works in other languages. Due to the risk of ambiguity (e.g. with a hotstring ending in */), any */ which is not preceded by /* is no longer ignored (reversing a change made in AHK_L revision 54).

Integer constants and numeric strings outside of the supported range (of 64-bit signed integers) now overflow/wrap around, instead of being capped at the min/max value. This is consistent with math operators, so 9223372036854775807+1 == 9223372036854775808 (but both produce -9223372036854775808). This facilitates bitwise operations on 64-bit values.

For numeric strings, there are fewer cases where whitespace characters other than space and tab are allowed to precede the number. The general rule (in both v1 and v2) is that only space and tab are permitted, but in some cases other whitespace characters are tolerated due to C runtime library conventions.

else can now be used with loop, for, while and catch. For loops, it is executed if the loop had zero iterations. For catch, it is executed if no exception is thrown within try (and is not executed if any error or value is thrown, even if there is no catch matching the value's class). Consequently, the interpretation of else may differ from previous versions when used without braces. For example:

```
if condition
{
   while condition
      ; statement to execute for each iteration
} ; These braces are now required, otherwise else associates with while
else
   ; statement to execute if condition is false
```

Continuation Sections

Smart LTrim: The default behaviour is to count the number of leading spaces or tabs on the first line below the continuation section options, and remove that many spaces or tabs from each line thereafter. If the first line mixes spaces and tabs, only the first type of character is treated as indentation. If any line is indented less than the first line or with the wrong characters, all leading whitespace on that line is left as is.

Quote marks are automatically escaped (i.e. they are interpreted as literal characters) if the continuation section starts inside a quoted string. This avoids the need to escape quote marks in multi-line strings (if the starting and ending quotes are outside the continuation section) while still allowing multi-line expressions to contain quoted strings.

If the line above the continuation section ends with a name character and the section does not start inside a quoted string, a single space is automatically inserted to separate the name from the contents of the continuation section. This allows a continuation section to be used for a multi-line expression following return, function call statements, etc. It also ensures variable names are not joined with other tokens (or names), causing invalid expressions.

Newline characters (`n) in expressions are treated as spaces. This allows multi-line expressions to be written using a continuation section with default options (i.e. omitting Join).

The , and % options have been removed, since there is no longer any need to escape these characters.

If (or) appears in the options of a potential continuation section (other than as part of the Join option), the overall line is not interpreted as the start of a continuation section. In other words, lines like (x.y)() and (x=y) && z() are interpreted as expressions. A multi-line expression can also begin with an open-parenthesis at the start of a line, provided that there is at least one other (or) on the first physical line. For example, the entire expression could be enclosed with ((...)).

Excluding the above case, if any invalid options are present, a load-time error is displayed instead of ignoring the invalid options.

Lines starting with (and ending with : are no longer excluded from starting a continuation section on the basis of looking like a label, as (is no longer valid in a label name. This makes it possible for something like (Join: to start a continuation section. However, (: is an error and (:: is still a hotkey.

A new method of line continuation is supported in expressions and function/property definitions which utilizes the fact that each (/[/{ must be matched with a corresponding)/]/}. In other words, if a line contains an unclosed (/[/{, it will be joined with subsequent lines until the number of opening and closing symbols balances out. Brace { at the end of a line is considered to be one-true-brace (rather than the start of an object literal) if there are no other unclosed symbols and the brace is not immediately preceded by an operator.

Continuation Lines

Line continuation is now more selective about the context in which a symbol is considered an expression operator. In general, comma and expression operators can no longer be used for continuation in a textual context, such as with hotstrings or directives (other than #HotIf), or after an unclosed quoted string.

Line continuation now works for expression operators at the end of a line.

is, in and contains are usable for line continuation, though in and contains are still reserved/not yet implemented as operators.

and, or, is, in and contains act as line continuation operators even if followed by an assignment or other binary operator, since these are no longer valid variable names. By contrast, v1 had exceptions for and/or followed by any of: <>=/|^:,

When . is used for continuation, the two lines are no longer automatically delimited by a space if there was no space or tab to the right of . at the start of a line, as in .VeryLongNestedClassName. Note that x .123 is always property access (not auto-concat) and x+.123 works with or without space.

Types

In general, v2 produces more consistent results with any code that depends on the type of a value.

In v1, a variable can contain both a string and a cached binary number, which is updated whenever the variable is used as a number. Since this cached binary number is the only means of detecting the type of value, caching performed internally by expressions like var+1 or abs(var) effectively changes the "type" of var as a side-effect. v2 disables this caching, so that str := "123" is always a string and int := 123 is always an integer. Consequently, str needs to be converted every time it is used as a number (instead of just the first time), unless it was originally assigned a pure number.

The built-in "variables" true, false, A_PtrSize, A_IsUnicode, A_Index and A_EventInfo always return pure integers, not strings. They sometimes return strings in v1 due to certain optimizations which have been superseded in v2.

All literal numbers are converted to pure binary numbers at load time and their string representation is discarded. For example, MsgBox 0x1 is equivalent to MsgBox 1, while MsgBox 1.0000 is equivalent to MsgBox 1.0 (because the float formatting has changed). Storing a number in a variable or returning it from a user-defined function retains its pure numeric status.

The default format specifier for floating-point numbers is now .17g (was 0.6f), which is more compact and more accurate in many cases. The default cannot be changed, but Format can be used to get different formatting.

Quoted literal strings and strings produced by concatenating with quoted literal strings are no longer unconditionally considered non-numeric. Instead, they are treated the same as strings stored in variables or returned from functions. This has the following implications:

Quoted literal "0" is considered false.

("0xA") + 1 and ("0x" Chr(65)) + 1 produce 11 instead of failing.

x[y:="0"] and x["0"] now behave the same.

The operators = and != now compare their operands alphabetically if both are strings, even if they are numeric strings. Numeric comparison is still performed when both operands are numeric and at least one operand is a pure number (not a string). So for example, 54 and "530" are compared numerically, while "54" and "530" are compared alphabetically. Additionally, strings stored in variables are treated no differently from literal strings.

The relational operators <, <=, > and >= now throw an exception if used with a non-numeric string. Previously they compared numerically or alphabetically depending on whether both inputs were numeric, but literal quoted strings were always considered non-numeric. Use StrCompare(a, b, CaseSense) instead.

Type(Value) returns one of the following strings: String, Integer, Float, or the specific class of an object.

Float(v), Integer(v) and String(v) convert v to the respective type, or throw an exception if the conversion cannot be performed (e.g. Integer("1z")). Number(v) converts to Integer or Float. String calls v.ToString() if v is an object. (Ideally this would be done for any implicit conversion from object to string, but the current implementation makes this difficult.)

Objects

Objects now use a more structured class-prototype approach, separating class/static members from instance members. Many of the built-in methods and Obj functions have been moved, renamed, changed or removed.

Each user-defined or built-in class is a class object (an instance of Class) exposing only methods and properties defined with the static keyword (including static members inherited from the base class) and nested classes.

Each class object has a Prototype property which becomes the base of all instances of that class. All non-static method and property definitions inside the class body are attached to the prototype object.

Instantiation is performed by calling the static Call method, as in myClass.Call() or myClass(). This allows the class to fully override construction behaviour (e.g. to implement a class factory or singleton, or to construct a native Array or Map instead of an Object), although initialization should still typically be performed in __New. The return value of __New is now ignored; to override the return value, do so from the Call method.

The mixed Object type has been split into Object, Array and Map (associative array).

Object is now the root class for all user-defined and built-in objects (this excludes VarRef and COM objects). Members added to Object.Prototype are inherited by all AutoHotkey objects.

The operator is expects a class, so x is y checks for y.Prototype in the base object chain. To check for y itself, call x.HasBase(y) or HasBase(x, y).

User-defined classes can also explicitly extend Object, Array, Map or some other built-in class (though doing so is not always useful), with Object being the default base class if none is specified.

The new operator has been removed. Instead, just omit the operator, as in MyClass(). To construct an object based on another object that is not a class, create it with {} or Object() (or by any other means) and set its base. __Init and __New can be called explicitly if needed, but generally this is only appropriate when instantiating a class.

Nested class definitions now produce a dynamic property with get and call accessor functions instead of a simple value property. This is to support the following behaviour:

Nested.Class() does not pass Nested to Nested.Class.Call and ultimately __New, which would otherwise happen because this is the normal behaviour for function objects called as methods (which is how the nested class is being used here).

Nested.Class := 1 is an error by default (the property is read-only).

Referring to or calling the class for the first time causes it to be initialized.

GetCapacity and SetCapacity were removed.

ObjGetCapacity and ObjSetCapacity now only affect the object's capacity to contain properties, and are not expected to be commonly used. Setting the capacity of the string buffer of a property, array element or map element is not supported; for binary data, use a Buffer object.

Array and Map have a Capacity property which corresponds to the object's current array or map allocation.

Other redundant Obj functions (which mirror built-in methods of Object) were removed. ObjHasOwnProp (formerly ObjHasKey) and ObjOwnProps (formerly ObjNewEnum) are kept to facilitate safe inspection of objects which have redefined those methods (and the primitive

prototypes, which don't have them defined). ObjCount was replaced with ObjOwnPropCount (a function only, for all Objects) and Map has its own Count property.

ObjRawGet and ObjRawSet were merged into GetOwnPropDesc and DefineProp. The original reasons for adding them were superseded by other changes, such as the Map type, changes to how meta-functions work, and DefineProp itself superseding meta-functions for some purposes.

Top-level class definitions now create a constant (read-only variable); that is, assigning to a class name is now an error rather than an optional warning, except where a local variable shadows the global class (which now occurs by default when assigning inside a function).

Primitive Values

Primitive values emulate objects by delegating method and property calls to a prototype object based on their type, instead of the v1 "default base object". Integer and Float extend Number. String and Number extend Primitive. Primitive and Object extend Any. These all exist as predefined classes.

Properties and Methods

Methods are defined by properties, unlike v2.0-a104 to v2.0-a127, where they are separate to properties. However, unlike v1, properties created by a class method definition (or built-in method) are read-only by default. Methods can still be created by assigning new value properties, which generally act as in v1.

The Object class defines new methods for dealing with properties and methods: DefineProp, DeleteProp, GetOwnPropDesc, HasOwnProp, OwnProps. Additional methods are defined for all values (except ComObjects): GetMethod, HasProp, HasMethod.

Object, Array and Map are now separate types, and array elements are separate from properties.

All built-in methods and properties (including base) are defined the same way as if user-defined. This ensures consistent behaviour and permits both built-in and user-defined members to be detected, retrieved or redefined.

If a property does not accept parameters, they are automatically passed to the object returned by the property (or it throws).

Attempting to retrieve a non-existent property is treated as an error for all types of values or objects, unless __get is defined. However, setting a non-existent property will create it in most cases.

Multi-dimension array hacks were removed. x.y[z]:=1 no longer creates an object in x.y, and x[y,z] is an error unless x.__item handles two parameters (or x.__item.__item does, etc.).

If a property defines get but not set, assigning a value throws instead of overriding the property.

DefineProp can be used to define what happens when a specific property is retrieved, set or called, without having to define any meta-functions. Property and method definitions in classes utilize the same mechanism, so it is possible to define a property getter/setter and a method with the same name.

{} object literals now directly set own property values or the object's base. That is, __Set and property setters are no longer invoked (which would typically only be possible if base is set within the parameter list).

Static/Class Variables

Static/class variable initializers are now executed within the context of a static __Init method, so this refers to the class and the initializers can create local variables. They are evaluated when the class is referenced for the first time (rather than being evaluated before the auto-execute section begins, strictly in the order of definition). If the class is not referenced sooner, they are evaluated when the class definition is reached during execution, so initialization of global variables can occur first, without putting them into a class.

Meta-Functions

Meta-functions were greatly simplified; they act like normal methods:

Where they are defined within the hierarchy is not important.

If overridden, the base version is not called automatically. Scripts can call super.__xxx() if needed.

If defined, it must perform the default action; e.g. if __set does not store the value, it is not stored.

Behaviour is not dependent on whether the method uses return (but of course, __get and __call still need to return a value).

Method and property parameters are passed as an Array. This optimizes for chained base/superclass calls and (in combination with MaxParams validation) encourages authors to handle the args. For __set, the value being assigned is passed separately.

this.__call(name, args)

this.__get(name, args)

this.__set(name, args, value)

Defined properties and methods take precedence over meta-functions, regardless of whether they were defined in a base object.

__Call is not called for internal calls to __Enum (formerly _NewEnum) or Call, such as when an object is passed to a for-loop or a function object is being called by SetTimer.

The static method __New is called for each class when it is initialized, if defined by that class or inherited from a superclass. See Static/Class Variables (above) and Class Initialization for more detail.

Array

class Array extends Object

An Array object contains a list or sequence of values, with index 1 being the first element.

When assigning or retrieving an array element, the absolute value of the index must be between 1 and the Length of the array, otherwise an exception is thrown. An array can be resized by inserting or removing elements with the appropriate method, or by assigning Length.

Currently brackets are required when accessing elements; i.e. a.1 refers to a property and a[1] refers to an element.

Negative values can be used to index in reverse.

Usage of Clone, Delete, InsertAt, Pop, Push and RemoveAt is basically unchanged. HasKey was renamed to Has. Length is now a property. The Capacity property was added.

Arrays can be constructed with Array(values*) or [values*]. Variadic functions receive an Array of parameters, and Arrays are also created by several built-in functions.

For-loop usage is for val in arr or for idx, val in arr, where idx = A_Index by default. That is, elements lacking a value are still enumerated, and the index is not returned if only one variable is passed.

Map

A Map object is an associative array with capabilities similar to the v1 Object, but less ambiguity.

Clone is used as before.

Delete can only delete one key at a time.

HasKey was renamed to Has.

Count is now a property.

New properties: Capacity, CaseSense

New methods: Get, Set, Clear

String keys are case-sensitive by default and are never converted to Integer.

Currently Float keys are still converted to strings.

Brackets are required when accessing elements; i.e. a.b refers to a property and a["b"] refers to an element. Unlike in v1, a property or method cannot be accidentally disabled by assigning an array element.

An exception is thrown if one attempts to retrieve the value of an element which does not exist, unless the map has a Default property defined. MapObj.Get(key, default) can be used to explicitly provide a default value for each request.

Use Map(Key, Value, ...) to create a map from a list of key-value pairs.

Enumeration

Changed enumerator model:

Replaced _NewEnum() with __Enum(n).

The required parameter n contains the number of variables in the for-loop, to allow it to affect enumeration without having to postpone initialization until the first iteration call.

Replaced Next() with Call(), with the same usage except that ByRef works differently now; for instance, a method defined as Call(&a) should assign a := next_value while Call(a) would receive a VarRef, so should assign %a% := next_value.

If __Enum is not present, the object is assumed to be an enumerator. This allows function objects (such as closures) to be used directly.

Since array elements and properties are now separate, enumerating properties requires explicitly creating an enumerator by calling OwnProps.

Bound Functions

When a bound function is called, parameters passed by the caller fill in any positions that were omitted when creating the bound function. For example, F.Bind(,b).Call(a,c) calls F(a,b,c) rather than F(,b,a,c).

COM Objects (ComObject)

COM wrapper objects now identify as instances of a few different classes depending on their variant type (which affects what methods and properties they support, as before):

ComValue is the base class for all COM wrapper objects.

ComObject is for VT_DISPATCH with a non-null pointer; that is, typically a valid COM object that can be invoked by the script using normal object syntax.

ComObjArray is for VT_ARRAY (SafeArrays).

ComValueRef is for VT_BYREF.

These classes can be used for type checks with obj is ComObject and similar. Properties and methods can be defined for objects of type ComValue, ComObjArray and ComValueRef (but not ComObject) by modifying the respective prototype object.

ComObject(CLSID) creates a ComObject; i.e. this is the new ComObjCreate.

Note: If you are updating old code and get a TypeError due to passing an Integer to ComObject, it's likely that you should be calling ComValue instead.

ComValue(vt, value) creates a wrapper object. It can return an instance of any of the classes listed above. This replaces ComObjParameter(vt, value), ComObject(vt, value) and any other names that were used with a variant type and value as parameters. value is converted to the appropriate type (following COM conventions), instead of requiring an integer with the right binary value. In particular, the following behave differently to before when passed an integer: R4, R8, Cy, Date. Pointer types permit either a pure integer address as before, or an object/ComValue.

ComObjFromPtr(pdsp) is a function similar to ComObjEnwrap(dsp), but like ObjFromPtr, it does not call AddRef on the pointer. The equivalent in v1 is ComObject(9, dsp, 1); omitting the third parameter in v1 caused an AddRef.

For both ComValue and ComObjFromPtr, be warned that AddRef is never called automatically; in that respect, they behave like ComObject(9, value, 1) or ComObject(13, value, 1) in v1. This does

not necessarily mean you should add ObjAddRef(value) when updating old scripts, as many scripts used the old function incorrectly.

COM wrapper objects with variant type VT_BYREF, VT_ARRAY or VT_UNKNOWN now have a Ptr property equivalent to ComObjValue(ComObj). This allows them to be passed to DllCall or ComCall with the Ptr arg type. It also allows the object to be passed directly to NumPut or NumGet, which may be used with VT_BYREF (access the caller's typed variable), VT_ARRAY (access SAFEARRAY fields) or VT_UNKNOWN (retrieve vtable pointer).

COM wrapper objects with variant type VT_DISPATCH or VT_UNKNOWN and a null interface pointer now have a Ptr property which can be read or assigned. Once assigned a non-null pointer, the property is read-only. This is intended for use with DllCall and ComCall, so the pointer does not need to be manually wrapped after the function returns.

Enumeration of ComObjArray is now consistent with Array; i.e. for value in arr or for index, value in arr rather than for value, vartype in arr. The starting value for index is the lower bound of the ComObjArray (arr.MinIndex()), typically 0.

The integer types I1, I8, UI1, UI2, UI4 and UI8 are now converted to Integer rather than String. These occur rarely in COM calls, but this also applies to VT_BYREF wrappers. VT_ERROR is no longer converted to Integer; it instead produces a ComValue.

COM objects no longer set A_LastError when a property or method invocation fails.

Default Property

A COM object may have a "default property", which has two uses:

The value of the object. For instance, in VBScript, MsgBox obj evaluates the object by invoking its default member.

The indexed property of a collection, which is usually named Item or item.

AutoHotkey v1 had no concept of a default property, so the COM object wrapper would invoke the default property if the property name was omitted; i.e. obj[] or obj[,x].

However, AutoHotkey v2 separates properties from array/map/collection items, and to do this obj[x] is mapped to the object's default property (whether or not x is present). For AutoHotkey objects, this is __Item.

Some COM objects which represent arrays or collections do not expose a default property, so items cannot be accessed with [] in v2. For instance, JavaScript array objects and some other objects normally used with JavaScript expose array elements as properties. In such cases, arr.%i% can be used to access an array element-property.

When an AutoHotkey v2 Array object is passed to JavaScript, its elements cannot be retrieved with JavaScript's arr[i], because that would attempt to access a property.

COM Calls

Calls to AutoHotkey objects via the IDispatch interface now transparently support VT_BYREF parameters. This would most commonly be used with COM events (ComObjConnect).

For each VT_BYREF parameter, an unnamed temporary var is created, the value is copied from the caller's variable, and a VarRef is passed to the AutoHotkey function/method. Upon return, the value is copied from the temporary var back into the caller's variable.

A function/method can assign a value by declaring the parameter ByRef (with &) or by explicit dereferencing.

For example, a parameter of type VT_BYREF|VT_BOOL would previously have received a ComObjRef object, and would be assigned a value like pbCancel[] := true or NumPut(-1, ComObjValue(pbCancel), "short"). Now the parameter can be defined as &bCancel and assigned like bCancel := true; or can be defined as pbCancel and assigned like %pbCancel% := true.

Library

Removed:

Asc() (use Ord)

AutoTrim (use Trim)

ComObjMissing() (write two consecutive commas instead)

ComObjUnwrap() (use ComObjValue instead, and ObjAddRef if needed)

ComObjEnwrap() (use ComObjFromPtr instead, and ObjAddRef if needed)

ComObjError()

ComObjXXX() where XXX is anything other than one of the explicitly defined ComObj functions (use ComObjActive, ComValue or ComObjFromPtr instead).

ControlSendRaw (use ControlSend "{Raw}" or ControlSendText instead)

EnvDiv

EnvMult

EnvUpdate (it is of very limited usefulness and can be replaced with a simple SendMessage)

Exception (use Error or an appropriate subclass)

FileReadLine (use a file-reading loop or FileOpen)

Func (use a direct reference like MyFunc)

Gosub

Gui, GuiControl, GuiControlGet (see Gui)

IfEqual

IfExist

IfGreater

IfGreaterOrEqual

IfInString

IfLess

IfLessOrEqual

IfMsgBox (MsgBox now returns the button name)

IfNotEqual

IfNotExist

IfNotInString

IfWinActive

IfWinExist

IfWinNotActive

IfWinNotExist

If between/is/in/contains (but see isXXX)

Input (use InputHook)

IsFunc

Menu (use the Menu/MenuBar class, TraySetIcon, A_IconTip, A_IconHidden and A_AllowMainWindow)

MenuGetHandle (use Menu.Handle)

MenuGetName (there are no menu names; MenuFromHandle is the closest replacement)

Progress (use Gui)

SendRaw (use Send "{Raw}" or SendText instead)

SetBatchLines (-1 is now the default behaviour)

SetEnv

SetFormat (Format can be used to format a string)

SoundGet/SoundSet (see Sound functions)

SoundGetWaveVolume/SoundSetWaveVolume (slightly different behaviour to SoundGet/SoundSet regarding balance, but neither one preserves balance)

SplashImage (use Gui)

SplashTextOn/Off (use Gui)

StringCaseSense (use various parameters)

StringGetPos (use InStr)

StringLeft

StringLen

StringMid

StringRight

StringTrimLeft

StringTrimRight -- use SubStr in place of these commands.

StringReplace (use StrReplace instead)

StringSplit (use StrSplit instead)

Transform

VarSetCapacity (use a Buffer object for binary data/structs and VarSetStrCapacity for UTF-16 strings)

WinGetActiveStats

WinGetActiveTitle

#CommentFlag

#Delimiter

#DerefChar

#EscapeChar

#HotkeyInterval (use A_HotkeyInterval)

#HotkeyModifierTimeout (use A_HotkeyModifierTimeout)

#IfWinActive, #IfWinExist, #IfWinNotActive, #IfWinNotExist (see #HotIf Optimization)

#InstallKeybdHook (use the InstallKeybdHook function)

#InstallMouseHook (use the InstallMouseHook function)

#KeyHistory (use KeyHistory N)

#LTrim

#MaxHotkeysPerInterval (use A_MaxHotkeysPerInterval)

#MaxMem (maximum capacity of each variable is now unlimited)

#MenuMaskKey (use A_MenuMaskKey)

#NoEnv (now default behaviour)

Renamed:

ComObjCreate() → ComObject, which is a class now

ComObjParameter() → ComValue, which is a class now

DriveSpaceFree → DriveGetSpaceFree

EnvAdd → DateAdd

EnvSub → DateDiff

FileCopyDir → DirCopy

FileCreateDir → DirCreate

FileMoveDir → DirMove

FileRemoveDir → DirDelete

FileSelectFile → FileSelect

FileSelectFolder → DirSelect

#If → #HotIf

#IfTimeout → HotIfTimeout

StringLower → StrLower and StrTitle

StringUpper → StrUpper and StrTitle

UrlDownloadToFile → Download

WinMenuSelectItem → MenuSelect

LV, TV and SB functions → methods of GuiControl

File.__Handle → File.Handle

Removed Commands (Details)

See above for the full list.

EnvUpdate was removed, but can be replaced with a simple call to SendMessage as follows:

SendMessage(0x1A, 0, StrPtr("Environment"), 0xFFFF)

StringCaseSense was removed, so != is always case-insensitive (but !== was added for case-sensitive not-equal), and both = and != only ignore case for ASCII characters. StrCompare was added for comparing strings using any mode. Various string functions now have a CaseSense parameter which can be used to specify case-sensitivity or the locale mode.

Modified Commands/Functions

About the section title: there are no commands in v2, just functions. The title refers to both versions.

BlockInput is no longer momentarily disabled whenever an Alt event is sent with the SendEvent method. This was originally done to work around a bug in some versions of Windows XP, where BlockInput blocked the artificial Alt event.

Chr(0) returns a string of length 1, containing a binary zero. This is a result of improved support for binary zero in strings.

ClipWait now returns 0 if the wait period expires, otherwise 1. ErrorLevel was removed. Specifying 0 is no longer the same as specifying 0.5; instead, it produces the shortest wait possible.

ComObj(): This function had a sort of wildcard name, allowing many different suffixes. Some names were more commonly used with specific types of parameters, such as ComObjActive(CLSID), ComObjParameter(vt, value), ComObjEnwrap(dsp). There are instead now separate functions/classes, and no more wildcard names. See COM Objects (ComObject) for details.

Control: Several changes have been made to the Control parameter used by the Control functions, SendMessage and PostMessage:

It can now accept a HWND (must be a pure integer) or an object with a Hwnd property, such as a GuiControl object. The HWND can identify a control or a top-level window, though the latter is usually only meaningful for a select few functions (see below).

It is no longer optional, except with functions which can operate on a top-level window (ControlSend[Text], ControlClick, SendMessage, PostMessage) or when preceded by other optional parameters (ListViewGetContent, ControlGetPos, ControlMove).

If omitted, the target window is used instead. This matches the previous behaviour of SendMessage/PostMessage, and replaces the ahk_parent special value previously used by ControlSend.

Blank values are invalid. Functions never default to the target window's topmost control.

ControlGetFocus now returns the control's HWND instead of its ClassNN, and no longer considers there to be an error when it has successfully determined that the window has no focused control.

ControlMove, ControlGetPos and ControlClick now use client coordinates (like GuiControl) instead of window coordinates. Client coordinates are relative to the top-left of the client area, which excludes the window's title bar and borders. (Controls are rendered only inside the client area.)

ControlMove, ControlSend and ControlSetText now use parameter order consistent with the other Control functions; i.e. Control, WinTitle, WinText, ExcludeTitle, ExcludeText are always grouped together (at the end of the parameter list), to aide memorisation.

CoordMode no longer accepts "Relative" as a mode, since all modes are relative to something. It was synonymous with "Window", so use that instead.

DllCall: See DllCall section further below.

Edit previously had fallback behaviour for the .ini file type if the "edit" shell verb was not registered. This was removed as script files are not expected to have the .ini extension. AutoHotkey.ini was the default script name in old versions of AutoHotkey.

Edit now does nothing if the script was read from stdin, instead of attempting to open an editor for *.

EnvSet now deletes the environment variable if the value parameter is completely omitted.

Exit previously acted as ExitApp when the script is not persistent, even if there were other suspended threads interrupted by the thread which called Exit. It no longer does this. Instead, it always exits the current thread properly, and (if non-persistent) the script terminates only after the last thread exits. This ensures finally statements are executed and local variables are freed, which may allow __delete to be called for any objects contained by local variables.

FileAppend defaults to no end-of-line translations, consistent with FileRead and FileOpen. FileAppend and FileRead both have a separate Options parameter which replaces the option prefixes and may include an optional encoding name (superseding FileRead's *Pnnn option). FileAppend, FileRead and FileOpen use "`n" to enable end-of-line translations. FileAppend and FileRead support an option "RAW" to disable codepage conversion (read/write binary data); FileRead returns a Buffer object in this case. This replaces *c (see ClipboardAll in the documentation). FileAppend may accept a Buffer-like object, in which case no conversions are performed.

FileCopy and FileMove now throw an exception if the source path does not contain * or ? and no file was not found. However, it is still not considered an error to copy or move zero files when the source path contains wildcards.

FileOpen now throws an exception if it fails to open the file. Otherwise, an exception would be thrown (if the script didn't check for failure) by the first attempt to access the object, rather than at the actual point of failure.

File.RawRead: When a variable is passed directly, the address of the variable's internal string buffer is no longer used. Therefore, a variable containing an address may be passed directly (whereas in v1, something like var+0 was necessary).

For buffers allocated by the script, the new Buffer object is preferred over a variable; any object can be used, but must have Ptr and Size properties.

File.RawWrite: As above, except that it can accept a string (or variable containing a string), in which case Bytes defaults to the size of the string in bytes. The string may contain binary zero.

File.ReadLine now always supports `r, `n and `r`n as line endings, and no longer includes the line ending in the return value. Line endings are still returned to the script as-is by File.Read if EOL translation is not enabled.

FileEncoding now allows code pages to be specified by number without the CP prefix. Its parameter is no longer optional, but can still be explicitly blank.

FileExist now ignores the . and .. implied in every directory listing, so FileExist("dir*") is now false instead of true when dir exists but is empty.

FileGetAttrib and A_LoopFileAttrib now include the letter "L" for reparse points or symbolic links.

FileInstall in a non-compiled script no longer attempts to copy the file if source and destination are the same path (after resolving relative paths, as the source is relative to A_ScriptDir, not A_WorkingDir). In v1 this caused ErrorLevel to be set to 1, which mostly went unnoticed. Attempting to copy a file onto itself via two different paths still causes an error.

FileSelectFile (now named FileSelect) had two multi-select modes, accessible via options 4 and M. Option 4 and the corresponding mode have been removed; they had been undocumented for some time. FileSelect now returns an Array of paths when the multi-select mode is used, instead of a string like C:\Dir`nFile1`nFile2. Each array element contains the full path of a file. If the user cancels, the array is empty.

FileSelect now uses the IFileDialog API present in Windows Vista and later, instead of the old GetOpenFileName/GetSaveFileName API. This removes the need for (built-in) workarounds relating to the dialog changing the current working directory.

FileSelect no longer has a redundant "Text Documents (*.txt)" filter by default when Filter is omitted.

FileSelect no longer strips spaces from the filter pattern, such as for pattern with spaces*.ext. Testing indicates spaces on either side of the pattern (such as after the semi-colon in *.cpp; *.h) are already ignored by the OS, so there should be no negative consequences.

FileSelect can now be used in "Select Folder" mode via the D option letter.

FileSetAttrib now overwrites attributes when no +, - or ^ prefix is present, instead of doing nothing. For example, FileSetAttrib(FileGetAttrib(file2), file1) copies the attributes of file2 to file1 (adding any that file2 have and removing any that it does not have).

FileSetAttrib and FileSetTime: the OperateOnFolders? and Recurse? parameters have been replaced with a single Mode parameter identical to that of Loop Files. For example, FileSetAttrib("+a", "*.zip", "RF") (Recursively operate on Files only).

GetKeyName now returns the non-Numpad names for VK codes that correspond to both a Numpad and a non-Numpad key. For instance, GetKeyName("vk25") returns Left instead of NumpadLeft.

GetKeyState now always returns 0 or 1.

GroupActivate now returns the HWND of the window which was selected for activation, or 0 if there were no matches (aside from the already-active window), instead of setting ErrorLevel.

GroupAdd: Removed the Label parameter and related functionality. This was an unintuitive way to detect when GroupActivate fails to find any matching windows; GroupActivate's return value should be used instead.

GroupDeactivate now selects windows in a manner closer to the Alt+Esc and Alt+Shift+Esc system hotkeys and the taskbar. Specifically,

Owned windows are not evaluated. If the owner window is eligible (not a match for the group), either the owner window or one of its owned windows is activated; whichever was active last. A window owned by a group member will no longer be activated, but adding the owned window itself to the group now has no effect. (The previous behaviour was to cycle through every owned window and never activate the owner.)

Any disabled window is skipped, unless one of its owned windows was active more recently than it.

Windows with the WS_EX_NOACTIVATE style are skipped, since they are probably not supposed to be activated. They are also skipped by the Alt+Esc and Alt+Shift+Esc system hotkeys.

Windows with WS_EX_TOOLWINDOW but not WS_EX_APPWINDOW are omitted from the taskbar and Alt-Tab, and are therefore skipped.

Hotkey no longer defaults to the script's bottommost #HotIf (formerly #If). Hotkey/hotstring and HotIf threads default to the same criterion as the hotkey, so Hotkey A_ThisHotkey, "Off" turns off the current hotkey even if it is context-sensitive. All other threads default to the last setting used by the auto-execute section, which itself defaults to no criterion (global hotkeys).

Hotkey's Callback parameter now requires a function object or hotkey name. Labels and function names are no longer supported. If a hotkey name is specified, the original function of that hotkey is used; and unlike before, this works with #HotIf (formerly #If).

Among other benefits, this eliminates ambiguity with the following special strings: On, Off, Toggle, AltTab, ShiftAltTab, AltTabAndMenu, AltTabMenuDismiss. The old behaviour was to use the label/function by that name if one existed, but only if the Label parameter did not contain a variable reference or expression.

Hotkey and Hotstring now support the S option to make the hotkey/hostring exempt from Suspend (equivalent to the new #SuspendExempt directive), and the S0 option to disable exemption.

"Hotkey If" and the other If sub-commands were replaced with individual functions: HotIf, HotIfWinActive, HotIfWinExist, HotIfWinNotActive, HotIfWinNotExist.

HotIf (formerly "Hotkey If") now recognizes expressions which use the and or or operators. This did not work in v1 as these operators were replaced with && or || at load time.

Hotkey no longer has a UseErrorLevel option, and never sets ErrorLevel. An exception is thrown on failure. Error messages were changed to be constant (and shorter), with the key or hotkey name in Exception.Extra, and the class of the exception indicating the reason for failure.

#HotIf (formerly #If) now implicitly creates a function with one parameter (ThisHotkey). As is the default for all functions, this function is assume-local. The expression can create local variables and

read global variables, but cannot directly assign to global variables as the expression cannot contain declarations.

#HotIf has been optimized so that simple calls to WinActive or WinExist can be evaluated directly by the hook thread (as #IfWin was in v1, and HotIfWin still is). This improves performance and reduces the risk of problems when the script is busy/unresponsive. This optimization applies to expressions which contain a single call to WinActive or WinExist with up to two parameters, where each parameter is a simple quoted string and the result is optionally inverted with ! or not. For example, #HotIf WinActive("Chrome") or #HotIf !WinExist("Popup"). In these cases, the first expression with any given combination of criteria can be identified by either the expression or the window criteria. For example, HotIf '!WinExist("Popup")' and HotIfWinNotExist "Popup" refer to the same hotkey variants.

KeyHistory N resizes the key history buffer instead of displaying the key history. This replaces "#KeyHistory N".

ImageSearch returns true if the image was found, false if it was not found, or throws an exception if the search could not be conducted. ErrorLevel is not set.

IniDelete, IniRead and IniWrite set A_LastError to the result of the operating system's GetLastError() function.

IniRead throws an exception if the requested key, section or file cannot be found and the Default parameter was omitted. If Default is given a value, even "", no exception is thrown.

InputHook now treats Shift+Backspace the same as Backspace, instead of transcribing it to `b.

InputBox has been given a syntax overhaul to make it easier to use (with fewer parameters). See InputBox for usage.

InStr's CaseSensitive parameter has been replaced with CaseSense, which can be 0, 1 or "Locale".

InStr now searches right-to-left when Occurrence is negative (which previously caused a result of 0), and no longer searches right-to-left if a negative StartingPos is used with a positive Occurrence. (However, it still searches right-to-left if StartingPos is negative and Occurrence is omitted.) This facilitates right-to-left searches in a loop, and allows a negative StartingPos to be used while still searching left-to-right.

For example, InStr(a, b,, -1, 2) now searches left-to-right. To instead search right-to-left, use InStr(a, b,, -1, -2).

Note that a StartingPos of -1 means the last character in v2, but the second last character in v1. If the example above came from v1 (rather than v2.0-a033 - v2.0-a136), the new code should be InStr(a, b, -2, -2).

KeyWait now returns 0 if the wait period expires, otherwise 1. ErrorLevel was removed.

MouseClick and MouseClickDrag are no longer affected by the system setting for swapped mouse buttons; "Left" is the always the primary button and "Right" is the secondary.

MsgBox has had its syntax changed to prioritise its most commonly used parameters and improve ease of use. See MsgBox further below for a summary of usage.

NumPut/NumGet: When a variable is passed directly, the address of the variable's internal string buffer is no longer used. Therefore, a variable containing an address may be passed directly (whereas in v1, something like var+0 was necessary). For buffers allocated by the script, the new Buffer object is preferred over a variable; any object can be used, but must have Ptr and Size properties.

NumPut's parameters were reordered to allow a sequence of values, with the (now mandatory) type string preceding each number. For example: NumPut("ptr", a, "int", b, "int", c, addrOrBuffer, offset). Type is now mandatory for NumGet as well. (In comparison to DllCall, NumPut's input parameters correspond to the dll function's parameters, while NumGet's return type parameter corresponds to the dll function's return type string.)

The use of Object(obj) and Object(ptr) to convert between a reference and a pointer was shifted to separate functions, ObjPtrAddRef(obj) and ObjFromPtrAddRef(ptr). There are also versions of these functions that do not increment the reference count: ObjPtr(obj) and ObjFromPtr(ptr).

The OnClipboardChange label is no longer called automatically if it exists. Use the OnClipboardChange function which was added in v1.1.20 instead. It now requires a function object, not a name.

OnError now requires a function object, not a name. See also Error Handling further below.

The OnExit command has been removed; use the OnExit function which was added in v1.1.20 instead. It now requires a function object, not a name. A_ExitReason has also been removed; its value is available as a parameter of the OnExit callback function.

OnMessage no longer has the single-function-per-message mode that was used when a function name (string) was passed; it now only accepts a function by reference. Use OnMessage(x, MyFunc) where MyFunc is literally the name of a function, but note that the v1 equivalent would be OnMessage(x, Func("MyFunc")), which allows other functions to continue monitoring message x, unlike OnMessage(x, "MyFunc"). To stop monitoring the message, use OnMessage(x, MyFunc, 0) as OnMessage(x, "") and OnMessage(x) are now errors. On failure, OnMessage throws an exception.

Pause is no longer exempt from #MaxThreadsPerHotkey when used on the first line of a hotkey, so #p::Pause is no longer suitable for toggling pause. Therefore, Pause() now only pauses the current thread (for combinations like ListVars/Pause), while Pause(v) now always operates on the underlying thread. v must be 0, 1 or -1. The second parameter was removed.

PixelSearch and PixelGetColor use RGB values instead of BGR, for consistency with other functions. Both functions throw an exception if a problem occurs, and no longer set ErrorLevel. PixelSearch returns true if the color was found. PixelSearch's slow mode was removed, as it is unusable on most modern systems due to an incompatibility with desktop composition.

PostMessage: See SendMessage further below.

Random has been reworked to utilize the operating system's random number generator, lift several restrictions, and make it more convenient to use.

The full 64-bit range of signed integer values is now supported (increased from 32-bit).

Floating-point numbers are generated from a 53-bit random integer, instead of a 32-bit random integer, and should be greater than or equal to Min and lesser than Max (but floating-point rounding errors can theoretically produce equal to Max).

The parameters could already be specified in any order, but now specifying only the first parameter defaults the other bound to 0 instead of 2147483647. For example, Random(9) returns a number between 0 and 9.

If both parameters are omitted, the return value is a floating-point number between 0.0 (inclusive) and 1.0 (generally exclusive), instead of an integer between 0 and 2147483647 (inclusive).

The system automatically seeds the random number generator, and does not provide a way to manually seed it, so there is no replacement for the NewSeed parameter.

RegExMatch options O and P were removed; O (object) mode is now mandatory. The RegExMatch object now supports enumeration (for-loop). The match object's syntax has changed:

__Get is used to implement the shorthand match.subpat where subpat is the name of a subpattern/capturing group. As __Get is no longer called if a property is inherited, the following subpattern names can no longer be used with the shorthand syntax: Pos, Len, Name, Count, Mark. (For example, match.Len always returns the length of the overall match, not a captured string.)

Originally the match object had methods instead of properties so that properties could be reserved for subpattern names. As new language behaviour implies that match.name would return a function by default, the methods have been replaced or supplemented with properties:

Pos, Len and Name are now properties and methods.

Name now requires 1 parameter to avoid confusion (match.Name throws an error).

Count and Mark are now only properties.

Value has been removed; use match.0 or match[] instead of match.Value(), and match[N] instead of match.Value(N).

RegisterCallback was renamed to CallbackCreate and changed to better utilize closures:

It now supports function objects (and no longer supports function names).

Removed EventInfo parameter (use a closure or bound function instead).

Removed the special behaviour of variadic callback functions and added the & option (pass the address of the parameter list).

Added CallbackFree(Address), to free the callback memory and release the associated function object.

Registry functions (RegRead, RegWrite, RegDelete): the new syntax added in v1.1.21+ is now the only syntax. Root key and subkey are combined. Instead of RootKey, Key, write RootKey\Key. To connect to a remote registry, use \\ComputerName\RootKey\Key instead of \\ComputerName:RootKey, Key.

RegWrite's parameters were reordered to put Value first, like IniWrite (but this doesn't affect the single-parameter mode, where Value was the only parameter).

When KeyName is omitted and the current loop reg item is a subkey, RegDelete, RegRead and RegWrite now operate on values within that subkey; i.e. KeyName defaults to A_LoopRegKey "\" A_LoopRegName in that case (note that A_LoopRegKey was merged with A_LoopRegSubKey). Previously they behaved as follows:

RegRead read a value with the same name as the subkey, if one existed in the parent key.

RegWrite returned an error.

RegDelete deleted the subkey.

RegDelete, RegRead and RegWrite now allow ValueName to be specified when KeyName is omitted:

If the current loop reg item is a subkey, ValueName defaults to empty (the subkey's default value) and ValueType must be specified.

If the current loop reg item is a value, ValueName and ValueType default to that value's name and type, but one or both can be overridden.

Otherwise, RegDelete with a blank or omitted ValueName now deletes the key's default value (not the key itself), for consistency with RegWrite, RegRead and A_LoopRegName. The phrase "AHK_DEFAULT" no longer has any special meaning. To delete a key, use RegDeleteKey (new).

RegRead now has a Default parameter, like IniRead.

RegRead had an undocumented 5-parameter mode, where the value type was specified after the output variable. This has been removed.

Reload now does nothing if the script was read from stdin.

Run and RunWait no longer recognize the UseErrorLevel option as ErrorLevel was removed. Use try/catch instead. A_LastError is set unconditionally, and can be inspected after an exception is caught/suppressed. RunWait returns the exit code.

Send (and its variants) now interpret {LButton} and {RButton} in a way consistent with hotkeys and Click. That is, LButton is the primary button and RButton is the secondary button, even if the user has swapped the buttons via system settings.

SendMessage and PostMessage now require wParam and lParam to be integers or objects with a Ptr property; an exception is thrown if they are given a non-numeric string or float. Previously a string was passed by address if the expression began with ", but other strings were coerced to integers. Passing the address of a variable (formerly &var, now StrPtr(var)) no longer updates the variable's length (use VarSetStrCapacity(&var, -1)).

SendMessage and PostMessage now throw an exception on failure (or timeout) and do not set ErrorLevel. SendMessage returns the message reply.

SetTimer no longer supports label or function names, but as it now accepts an expression and functions can be referenced directly by name, usage looks very similar: SetTimer MyFunc. As with all other functions which accept an object, SetTimer now allows expressions which return an object (previously it required a variable reference).

Sort has received the following changes:

The VarName parameter has been split into separate input/output parameters, for flexibility. Usage is now Output := Sort(Input [, Options, Function]).

When any two items compare equal, the original order of the items is now automatically used as a tie-breaker to ensure more stable results.

The C option now also accepts a suffix equivalent to the CaseSense parameter of other functions (in addition to CL): CLocale CLogical COn C1 COff C0. In particular, support for the "logical" comparison mode is new.

Sound functions: SoundGet and SoundSet have been revised to better match the capabilities of the Vista+ sound APIs, dropping support for XP.

Removed unsupported control types.

Removed legacy mixer component types.

Let components be referenced by name and/or index.

Let devices be referenced by name-prefix and/or index.

Split into separate Volume and Mute functions.

Added SoundGetName for retrieving device or component names.

Added SoundGetInterface for retrieving COM interfaces.

StrGet: If Length is negative, its absolute value indicates the exact number of characters to convert, including any binary zeros that the string might contain -- in other words, the result is always a string of exactly that length. If Length is positive, the converted string ends at the first binary zero as in v1.

StrGet/StrPut: The Address parameter can be an object with the Ptr and Size properties, such as the new Buffer object. The read/write is automatically limited by Size (which is in bytes). If Length is also specified, it must not exceed Size (multiplied by 2 for UTF-16).

StrPut's return value is now in bytes, so it can be passed directly to Buffer().

StrReplace now has a CaseSense parameter in place of OutputVarCount, which is moved one parameter to the right, with Limit following it.

Suspend: Making a hotkey or hotstring's first line a call to Suspend no longer automatically makes it exempt from suspension. Instead, use #SuspendExempt or the S option. The "Permit" parameter value is no longer valid.

Switch now performs case-sensitive comparison for strings by default, and has a CaseSense parameter which overrides the mode of case sensitivity and forces string (rather than numeric) comparison. Previously it was case-sensitive only if StringCaseSense was changed to On.

SysGet now only has numeric sub-commands; its other sub-commands have been split into functions. See Sub-Commands further below for details.

TrayTip's usage has changed to TrayTip [Text, Title, Options]. Options is a string of zero or more case-insensitive options delimited by a space or tab. The options are Iconx, Icon!, Iconi, Mute and/or any numeric value as before. TrayTip now shows even if Text is omitted (which is now harder to do by accident than in v1). The Seconds parameter no long exists (it had no effect on Windows Vista or later). Scripts may now use the NIIF_USER (0x4) and NIIF_LARGE_ICON (0x20) flags in combination (0x24) to include the large version of the tray icon in the notification. NIIF_USER (0x4) can also be used on its own for the small icon, but may not have consistent results across all OSes.

#Warn UseUnsetLocal and UseUnsetGlobal have been removed, as reading an unset variable now raises an error. IsSet can be used to avoid the error and try/catch or OnError can be used to handle it.

#Warn VarUnset was added; it defaults to MsgBox. If not disabled, a warning is given for the first non-dynamic reference to each variable which is never used as the target of a direct, non-dynamic assignment or the reference operator (&), or passed directly to IsSet.

#Warn Unreachable no longer considers lines following an Exit call to be unreachable, as Exit is now an ordinary function.

#Warn ClassOverwrite has been removed, as top-level classes can no longer be overwritten by assignment. (However, they can now be implicitly shadowed by a local variable; that can be detected by #Warn LocalSameAsGlobal.)

WinActivate now sends {Alt up} after its first failed attempt at activating a window. Testing has shown this reduces the occurrence of flashing taskbar buttons. See the documentation for more details.

WinSetTitle and WinMove now use parameter order consistent with other Win functions; i.e. WinTitle, WinText, ExcludeTitle, ExcludeText are always grouped together (at the end of the parameter list), to aide memorisation.

The WinTitle parameter of various functions can now accept a HWND (must be a pure integer) or an object with a Hwnd property, such as a Gui object. DetectHiddenWindows is ignored in such cases.

WinMove no longer has special handling for the literal word DEFAULT. Omit the parameter or specify an empty string instead (this works in both v1 and v2).

WinWait, WinWaitClose, WinWaitActive and WinWaitNotActive return non-zero if the wait finished (timeout did not expire). ErrorLevel was removed. WinWait and WinWaitActive return the HWND of the found window. WinWaitClose now sets the Last Found Window, so if WinWaitClose times out, it returns false and WinExist() returns the last window it found. For the timeout, specifying 0 is no longer the same as specifying 0.5; instead, it produces the shortest wait possible.

Unsorted:

A negative StartingPos for InStr, SubStr, RegExMatch and RegExReplace is interpreted as a position from the end. Position -1 is the last character and position 0 is invalid (whereas in v1, position 0 was the last character).

Functions which previously accepted On/Off or On/Off/Toggle (but not other strings) now require 1/0/-1 instead. On and Off would typically be replaced with True and False. Variables which returned On/Off now return 1/0, which are more useful in expressions.

#UseHook and #MaxThreadsBuffer allow 1, 0, True and False. (Unlike the others, they do not actually support expressions.)

ListLines allows blank or boolean.

ControlSetChecked, ControlSetEnabled, Pause, Suspend, WinSetAlwaysOnTop, and WinSetEnabled allow 1, 0 and -1.

A_DetectHiddenWindows, A_DetectHiddenText, and A_StoreCapsLockMode use boolean (as do the corresponding functions).

The following functions return a pure integer instead of a hexadecimal string:

ControlGetExStyle

ControlGetHwnd

ControlGetStyle

MouseGetPos

WinActive

WinExist

WinGetID

WinGetIDLast

WinGetList (within the Array)

WinGetStyle

WinGetStyleEx

WinGetControlsHwnd (within the Array)

A_ScriptHwnd also returns a pure integer.

DllCall

If a type parameter is a variable, that variable's content is always used, never its name. In other words, unquoted type names are no longer supported - type names must be enclosed in quote marks.

When DllCall updates the length of a variable passed as Str or WStr, it now detects if the string was not properly null-terminated (likely indicating that buffer overrun has occurred), and terminates the program with an error message if so, as safe execution cannot be guaranteed.

AStr (without any suffix) is now input-only. Since the buffer is only ever as large as the input string, it was usually not useful for output parameters. This would apply to WStr instead of AStr if AutoHotkey is compiled for ANSI, but official v2 releases are only ever compiled for Unicode.

If a function writes a new address to a Str*, AStr* or WStr* parameter, DllCall now assigns the new string to the corresponding variable if one was supplied, instead of merely updating the length of the original string (which probably hasn't changed). Parameters of this type are usually not used to modify the input string, but rather to pass back a string at a new address.

DllCall now accepts an object for any Ptr parameter and the Function parameter; the object must have a Ptr property. For buffers allocated by the script, the new Buffer object is preferred over a variable. For Ptr*, the parameter's new value is assigned back to the object's Ptr property. This allows constructs such as DllCall(..., "Ptr*", unk := IUnknown.new()), which reduces repetition compared to DllCall(..., "Ptr*", punk), unk := IUnknown.new(punk), and can be used to ensure any output from the function is properly freed (even if an exception is thrown due to the HRESULT return type, although typically the function would not output a non-null pointer in that case).

DllCall now requires the values of numeric-type parameters to be numeric, and will throw an exception if given a non-numeric or empty string. In particular, if the * or P suffix is used for output parameters, the output variable is required to be initialized.

The output value (if any) of numeric parameters with the * or P suffix is ignored if the script passes a plain variable containing a number. To receive the output value, pass a VarRef such as &myVar or an object with a Ptr property.

The new HRESULT return type throws an exception if the function failed (int < 0 or uint & 0x80000000). This should be used only with functions that actually return a HRESULT.

Loop Sub-commands

The sub-command keyword must be written literally; it must not be enclosed in quote marks and cannot be a variable or expression. All other parameters are expressions. All loop sub-commands now support OTB.

Removed:

Loop, FilePattern [, IncludeFolders?, Recurse?]

Loop, RootKey [, Key, IncludeSubkeys?, Recurse?]

Use the following (added in v1.1.21) instead:

Loop Files, FilePattern [, Mode]

Loop Reg, RootKey\Key [, Mode]

The comma after the second word is now optional.

A_LoopRegKey now contains the root key and subkey, and A_LoopRegSubKey was removed.

InputBox

Obj := InputBox([Text, Title, Options, Default])

The Options parameter accepts a string of zero or more case-insensitive options delimited by a space or tab, similar to Gui control options. For example, this includes all supported options: x0 y0 w100 h100 T10.0 Password*. T is timeout and Password has the same usage as the equivalent Edit control option.

The width and height options now set the size of the client area (the area excluding the title bar and window frame), so are less theme-dependent.

The title will be blank if the Title parameter is an empty string. It defaults to A_ScriptName only when completely omitted, consistent with optional parameters of user-defined functions.

Obj is an object with the properties result (containing "OK", "Cancel" or "Timeout") and value.

MsgBox

Result := MsgBox([Text, Title, Options])

The Options parameter accepts a string of zero or more case-insensitive options delimited by a space or tab, similar to Gui control options.

Iconx, Icon?, Icon! and Iconi set the icon.

Default followed immediately by an integer sets the nth button as default.

T followed immediately by an integer or floating-point number sets the timeout, in seconds.

Owner followed immediately by a HWND sets the owner, overriding the Gui +OwnDialogs option.

One of the following mutually-exclusive strings sets the button choices: OK, OKCancel, AbortRetryIgnore, YesNoCancel, YesNo, RetryCancel, CancelTryAgainContinue, or just the initials separated by slashes (o/c, y/n, etc.), or just the initials without slashes.

Any numeric value, the same as in v1. Numeric values can be combined with string options, or Options can be a pure integer.

The return value is the name of the button, without spaces. These are the same strings that were used with IfMsgBox in v1.

The title will be blank if the Title parameter is an empty string. It defaults to A_ScriptName only when completely omitted, consistent with optional parameters of user-defined functions.

Sub-Commands

Sub-commands of Control, ControlGet, Drive, DriveGet, WinGet, WinSet and Process have been replaced with individual functions, and the main commands have been removed. Names and usage have been changed for several of the functions. The new usage is shown below:

; Where ... means optional Control, WinTitle, etc.

```
Bool  := ControlGetChecked(...)
Bool  := ControlGetEnabled(...)
Bool  := ControlGetVisible(...)
Int   := ControlGetIndex(...)  ; For Tab, LB, CB, DDL
Str   := ControlGetChoice(...)
Arr   := ControlGetItems(...)
Int   := ControlGetStyle(...)
Int   := ControlGetExStyle(...)
Int   := ControlGetHwnd(...)

      ControlSetChecked(TrueFalseToggle, ...)
      ControlSetEnabled(TrueFalseToggle, ...)
      ControlShow(...)
      ControlHide(...)
      ControlSetStyle(Value, ...)
      ControlSetExStyle(Value, ...)
      ControlShowDropDown(...)
      ControlHideDropDown(...)
      ControlChooseIndex(Index, ...)  ; Also covers Tab
Index := ControlChooseString(Str, ...)

Index := ControlFindItem(Str, ...)
Index := ControlAddItem(Str, ...)
      ControlDeleteItem(Index, ...)

Int   := EditGetLineCount(...)
```

```
Int  := EditGetCurrentLine(...)
Int  := EditGetCurrentCol(...)
Str  := EditGetLine(N [, ...])
Str  := EditGetSelectedText(...)
        EditPaste(Str, ...)

Str  := ListViewGetContent([Options, ...])

        DriveEject([Drive])
        DriveRetract([Drive])
        DriveLock(Drive)
        DriveUnlock(Drive)
        DriveSetLabel(Drive [, Label])

Str  := DriveGetList([Type])
Str  := DriveGetFilesystem(Drive)
Str  := DriveGetLabel(Drive)
Str  := DriveGetSerial(Drive)
Str  := DriveGetType(Path)
Str  := DriveGetStatus(Path)
Str  := DriveGetStatusCD(Drive)
Int  := DriveGetCapacity(Path)
Int  := DriveGetSpaceFree(Path)

; Where ... means optional WinTitle, etc.
```

Int := WinGetID(...)

Int := WinGetIDLast(...)

Int := WinGetPID(...)

Str := WinGetProcessName(...)

Str := WinGetProcessPath(...)

Int := WinGetCount(...)

Arr := WinGetList(...)

Int := WinGetMinMax(...)

Arr := WinGetControls(...)

Arr := WinGetControlsHwnd(...)

Int := WinGetTransparent(...)

Str := WinGetTransColor(...)

Int := WinGetStyle(...)

Int := WinGetExStyle(...)

WinSetTransparent(N [, ...])

WinSetTransColor("Color [N]" [, ...]),

WinSetAlwaysOnTop([TrueFalseToggle := -1, ...])

WinSetStyle(Value [, ...])

WinSetExStyle(Value [, ...])

WinSetEnabled(Value [, ...])

WinSetRegion(Value [, ...])

WinRedraw(...)

WinMoveBottom(...)

WinMoveTop(...)

PID := ProcessExist([PID_or_Name])

PID := ProcessClose(PID_or_Name)

PID := ProcessWait(PID_or_Name [, Timeout])

PID := ProcessWaitClose(PID_or_Name [, Timeout])

ProcessSetPriority(Priority [, PID_or_Name])

ProcessExist, ProcessClose, ProcessWait and ProcessWaitClose no longer set ErrorLevel; instead, they return the PID.

None of the other functions set ErrorLevel. Instead, they throw an exception on failure. In most cases failure is because the target window or control was not found.

HWNDs and styles are always returned as pure integers, not hexadecimal strings.

ControlChooseIndex allows 0 to deselect the current item/all items. It replaces "Control Choose", but also supports Tab controls.

"ControlGet Tab" was merged into ControlGetIndex, which also works with ListBox, ComboBox and DDL. For Tab controls, it returns 0 if no tab is selected (rare but valid). ControlChooseIndex does not permit 0 for Tab controls since applications tend not to handle it.

ControlGetItems replaces "ControlGet List" for ListBox and ComboBox. It returns an Array.

DriveEject and DriveRetract now use DeviceIoControl instead of mciSendString. DriveEject is therefore able to eject non-CD/DVD drives which have an "Eject" option in Explorer (i.e. removable drives but not external hard drives which show as fixed disks).

ListViewGetContent replaces "ControlGet List" for ListView, and currently has the same usage as before.

WinGetList, WinGetControls and WinGetControlsHwnd return arrays, not newline-delimited lists.

WinSetTransparent treats "" as "Off" rather than 0 (which would make the window invisible and unclickable).

Abbreviated aliases such as Topmost, Trans, FS and Cap were removed.

The following functions were formerly sub-commands of SysGet:

Exists := MonitorGet(N, Left, Top, Right, Bottom)

Exists := MonitorGetWorkArea(N, Left, Top, Right, Bottom)

Count   := MonitorGetCount()

Primary := MonitorGetPrimary()

Name   := MonitorGetName(N)

New Functions

Buffer(Size, FillByte) (calling the Buffer class) creates and returns a Buffer object encapsulating a block of Size bytes of memory, initialized only if FillByte is specified. BufferObj.Ptr returns the address and BufferObj.Size returns or sets the size in bytes (reallocating the block of memory). Any object with Ptr and Size properties can be passed to NumPut, NumGet, StrPut, StrGet, File.RawRead, File.RawWrite and FileAppend. Any object with a Ptr property can be passed to DllCall parameters with Ptr type, SendMessage and PostMessage.

CaretGetPos([&X, &Y]) retrieves the current coordinates of the caret (text insertion point). This ensures the X and Y coordinates always match up, and there is no caching to cause unexpected behaviour (such as A_CaretX/Y returning a value that's not in the current CoordMode).

ClipboardAll([Data, Size]) creates an object containing everything on the clipboard (optionally accepting data previously retrieved from the clipboard instead of using the clipboard's current contents). The methods of reading and writing clipboard file data are different. The data format is the same, except that the data size is always 32-bit, so that the data is portable between 32-bit and 64-bit builds. See the v2 documentation for details.

ComCall(offset, comobj, ...) is equivalent to DllCall(NumGet(NumGet(comobj.ptr) + offset * A_Index), "ptr", comobj.ptr, ...), but with the return type defaulting to "hresult" rather than "int".

ComObject (formerly ComObjCreate) and ComObjQuery now return a wrapper object even if an IID is specified. ComObjQuery permits the first parameter to be any object with a Ptr property.

ControlGetClassNN returns the ClassNN of the specified control.

ControlSendText, equivalent to ControlSendRaw but using {Text} mode instead of {Raw} mode.

DirExist(Path), with usage similar to FileExist. Note that InStr(FileExist(Pattern), "D") only tells you whether the first matching file is a folder, not whether a folder exists.

Float(v): See Types further above.

InstallKeybdHook(Install := true, Force := false) and InstallMouseHook(Install := true, Force := false) replace the corresponding directives, for increased flexibility.

Integer(v): See Types further above.

isXXX: The legacy command "if var is type" has been replaced with a series of functions: isAlnum, isAlpha, isDigit, isFloat, isInteger, isLower, isNumber, isSpace, isUpper, isXDigit. With the exception of isFloat, isInteger and isNumber, an exception is thrown if the parameter is not a string, as implicit conversion to string may cause counter-intuitive results.

IsSet(var), IsSetRef(&var): Returns true if the variable has been assigned a value (even if that value is an empty string), otherwise false. If false, attempting to read the variable within an expression would throw an error.

Menu()/MenuBar() returns a new Menu/MenuBar object, which has the following members corresponding to v1 Menu sub-commands. Methods: Add, AddStandard, Check, Delete, Disable, Enable, Insert, Rename, SetColor, SetIcon, Show, ToggleCheck, ToggleEnable, Uncheck. Properties: ClickCount, Default, Handle (replaces MenuGetHandle). A_TrayMenu also returns a Menu object. There is no UseErrorLevel mode, no global menu names, and no explicitly deleting the menu itself (that happens when all references are released; the Delete method is equivalent to v1 DeleteAll). Labels are not supported, only function objects. The AddStandard method adds the standard menu items and allows them to be individually modified as with custom items. Unlike v1, the Win32 menu is destroyed only when the object is deleted.

MenuFromHandle(Handle) returns the Menu object corresponding to a Win32 menu handle, if it was created by AutoHotkey.

Number(v): See Types further above.

Persistent(Persist := true) replaces the corresponding directive, increasing flexibility.

RegDeleteKey("RootKey\SubKey") deletes a registry key. (RegDelete now only deletes values, except when omitting all parameters in a registry loop.)

SendText, equivalent to SendRaw but using {Text} mode instead of {Raw} mode.

StrCompare(str1, str2 [, CaseSense := false]) returns -1 (str1 is less than str2), 0 (equal) or 1 (greater than). CaseSense can be "Locale".

String(v): See Types further above.

StrPtr(str) returns the address of a string. Unlike address-of in v1, it can be used with literal strings and temporary strings.

SysGetIPAddresses() returns an array of IP addresses, equivalent to the A_IPAddress variables which have been removed. Each reference to A_IPAddress%N% retrieved all addresses but returned only one, so retrieving multiple addresses took exponentially longer than necessary. The returned array can have zero or more elements.

TraySetIcon([FileName, IconNumber, Freeze]) replaces "Menu Tray, Icon".

VarSetStrCapacity(&Var [, NewCapacity]) replaces the v1 VarSetCapacity, but is intended for use only with UTF-16 strings (such as to optimize repeated concatenation); therefore NewCapacity and the return value are in characters, not bytes.

VerCompare(A, B) compares two version strings using the same algorithm as #Requires.

WinGetClientPos([&X, &Y, &W, &H, WinTitle, ...]) retrieves the position and size of the window's client area, in screen coordinates.

New Directives

#DllLoad [FileOrDirName]: Loads a DLL or EXE file before the script starts executing.

Built-in Variables

A_AhkPath always returns the path of the current executable/interpreter, even when the script is compiled. Previously it returned the path of the compiled script if a BIN file was used as the base file, but v2.0 releases no longer include BIN files.

A_IsCompiled returns 0 instead of "" if the script has not been compiled.

A_OSVersion always returns a string in the format major.minor.build, such as 6.1.7601 for Windows 7 SP1. A_OSType has been removed as only NT-based systems are supported.

A_TimeSincePriorHotkey returns "" instead of -1 whenever A_PriorHotkey is "", and likewise for A_TimeSinceThisHotkey when A_ThisHotkey is blank.

All built-in "virtual" variables now have the A_ prefix (specifics below). Any predefined variables which lack this prefix (such as Object) are just global variables. The distinction may be important since it is currently impossible to take a reference to a virtual variable (except when passed directly to a built-in function); however, A_Args is not a virtual variable.

Built-in variables which return numbers now return them as an integer rather than a string.

Renamed:

A_LoopFileFullPath → A_LoopFilePath (returns a relative path if the Loop's parameter was relative, so "full path" was misleading)

A_LoopFileLongPath → A_LoopFileFullPath

Clipboard → A_Clipboard

Removed:

ClipboardAll (replaced with the ClipboardAll function)

ComSpec (use A_ComSpec)

ProgramFiles (use A_ProgramFiles)

A_AutoTrim

A_BatchLines

A_CaretX, A_CaretY (use CaretGetPos)

A_DefaultGui, A_DefaultListView, A_DefaultTreeView

A_ExitReason

A_FormatFloat

A_FormatInteger

A_Gui, A_GuiControl, A_GuiControlEvent, A_GuiEvent, A_GuiX, A_GuiY, A_GuiWidth, A_GuiHeight (all replaced with parameters of event handlers)

A_IPAddress1, A_IPAddress2, A_IPAddress3, A_IPAddress4 (use SysGetIPAddresses)

A_IsUnicode (v2 is always Unicode; it can be replaced with StrLen(Chr(0xFFFF)) or redefined with global A_IsUnicode := 1)

A_StringCaseSense

A_ThisLabel

A_ThisMenu, A_ThisMenuItem, A_ThisMenuItemPos (use the menu item callback's parameters)

A_LoopRegSubKey (A_LoopRegKey now contains the root key and subkey)

True and False (still exist, but are now only keywords, not variables)

Added:

A_AllowMainWindow (read/write; replaces "Menu Tray, MainWindow/NoMainWindow")

A_HotkeyInterval (replaces #HotkeyInterval)

A_HotkeyModifierTimeout (replaces #HotkeyModifierTimeout)

A_InitialWorkingDir (see Default Settings further below)

A_MaxHotkeysPerInterval (replaces #MaxHotkeysPerInterval)

A_MenuMaskKey (replaces #MenuMaskKey)

The following built-in variables can be assigned values:

A_ControlDelay

A_CoordMode..

A_DefaultMouseSpeed

A_DetectHiddenText (also, it now returns 1 or 0 instead of "On" or "Off")

A_DetectHiddenWindows (also, it now returns 1 or 0 instead of "On" or "Off")

A_EventInfo

A_FileEncoding (also, it now returns "CP0" in place of "", and allows the "CP" prefix to be omitted when assigning)

A_IconHidden

A_IconTip (also, it now always reflects the tooltip, even if it is default or empty)

A_Index: For counted loops, modifying this affects how many iterations are performed. (The global nature of built-in variables means that an Enumerator function could set the index to be seen by a For loop.)

A_KeyDelay

A_KeyDelayPlay

A_KeyDuration

A_KeyDurationPlay

A_LastError: Calls the Win32 SetLastError() function. Also, it now returns an unsigned value.

A_ListLines

A_MouseDelay

A_MouseDelayPlay

A_RegView

A_ScriptName: Changes the default dialog title.

A_SendLevel

A_SendMode

A_StoreCapsLockMode (also, it now returns 1 or 0 instead of "On" or "Off")

A_TitleMatchMode

A_TitleMatchModeSpeed

A_WinDelay

A_WorkingDir: Same as calling SetWorkingDir.

Built-in Objects

File objects now strictly require property syntax when invoking properties and method syntax when invoking methods. For example, File.Pos(n) is not valid. An exception is thrown if there are too few or too many parameters, or if a read-only property is assigned a value.

File.Tell() was removed.

Func.IsByRef() now works with built-in functions.

Gui

Gui, GuiControl and GuiControlGet were replaced with Gui() and Gui/GuiControl objects, which are generally more flexible, more consistent, and easier to use.

A GUI is typically not referenced by name/number (although it can still be named with GuiObj.Name). Instead, a GUI object (and window) is created explicitly by instantiating the Gui class, as in GuiObj := Gui(). This object has methods and properties which replace the Gui sub-commands. GuiObj.Add() returns a GuiControl object, which has methods and properties which replace the GuiControl and GuiControlGet commands. One can store this object in a variable, or use GuiObj["Name"] or GuiCtrlFromHwnd(hwnd) to retrieve the object. It is also passed as a parameter whenever an event handler (the replacement of a g-label) is called.

The usage of these methods and properties is not 1:1. Many parts have been revised to be more consistent and flexible, and to fix bugs or limitations.

There are no "default" GUIs, as the target Gui or control object is always specified. LV/TV/SB functions were replaced with methods (of the control object), making it much easier to use multiple ListViews/TreeViews.

There are no built-in variables containing information about events. The information is passed as parameters to the function/method which handles the event, including the source Gui or control.

Controls can still be named and be referenced by name, but it's just a name (used with GuiObj["Name"] and GuiObj.Submit()), not an associated variable, so there is no need to declare or

create a global or static variable. The value is never stored in a variable automatically, but is accessible via GuiCtrl.Value. GuiObj.Submit() returns a new associative array using the control names as keys.

The vName option now just sets the control's name to Name.

The +HwndVarName option has been removed in favour of GuiCtrl.Hwnd.

There are no more "g-labels" or labels/functions which automatically handle GUI events. The script must register for each event of interest by calling the OnEvent method of the Gui or GuiControl. For example, rather than checking if (A_GuiEvent = "I" && InStr(ErrorLevel, "F", true)) in a g-label, the script would register a handler for the ItemFocus event: MyLV.OnEvent("ItemFocus", MyFunction). MyFunction would be called only for the ItemFocus event. It is not necessary to apply AltSubmit to enable additional events.

Arrays are used wherever a pipe-delimited list was previously used, such as to specify the items for a ListBox when creating it, when adding items, or when retrieving the selected items.

Scripts can define a class which extends Gui and handles its own events, keeping all of the GUI logic self-contained.

Gui sub-commands

Gui New → Gui(). Passing an empty title (not omitting it) now results in an empty title, not the default title.

Gui Add → GuiObj.Add() or GuiObj.AddControlType(); e.g. GuiObj.Add("Edit") or GuiObj.AddEdit().

Gui Show → GuiObj.Show(), but it has no Title parameter. The title can be specified as a parameter of Gui() or via the GuiObj.Title property. The initial focus is still set to the first input-capable control with the WS_TABSTOP style (as per default message processing by the system), unless that's a Button control, in which case focus is now shifted to the Default button.

Gui Submit → GuiObj.Submit(). It works like before, except that Submit() creates and returns a new object which contains all of the "associated variables".

Gui Destroy → GuiObj.Destroy(). The object still exists (until the script releases it) but cannot be used. A new GUI must be created (if needed). The window is also destroyed when the object is deleted, but the object is "kept alive" while the window is visible.

Gui Font → GuiObj.SetFont(). It is also possible to set a control's font directly, with GuiCtrl.SetFont().

Gui Color → GuiObj.BackColor sets/returns the background color. ControlColor (the second parameter) is not supported, but all controls which previously supported it can have a background set by the +Background option instead. Unlike "Gui Color", GuiObj.BackColor does not affect Progress controls or disabled/read-only Edit, DDL, ComboBox or TreeView (with -Theme) controls.

Gui Margin → GuiObj.MarginX and GuiObj.MarginY properties.

Gui Menu → GuiObj.MenuBar sets/returns a MenuBar object created with MenuBar().

Gui Cancel/Hide/Minimize/Maximize/Restore → Gui methods of the same name.

Gui Flash → GuiObj.Flash(), but use false instead of Off.

Gui Tab → TabControl.UseTab(). Defaults to matching a prefix of the tab name as before. Pass true for the second parameter to match the whole tab name, but unlike the v1 "Exact" mode, it is case-insensitive.

Events

See Events (OnEvent) for details of all explicitly supported GUI and GUI control events.

The Size event passes 0, -1 or 1 (consistent with WinGetMinMax) instead of 0, 1 or 2.

The ContextMenu event can be registered for each control, or for the whole GUI.

The DropFiles event swaps the FileArray and Ctrl parameters, to be consistent with ContextMenu.

The ContextMenu and DropFiles events use client coordinates instead of window coordinates (Client is also the default CoordMode in v2).

The following control events were removed, but detecting them is a simple case of passing the appropriate numeric notification code (defined in the Windows SDK) to GuiCtrl.OnNotify(): K, D, d, A, S, s, M, C, E and MonthCal's 1 and 2.

Control events do not pass the event name as a parameter (GUI events never did).

Custom's N and Normal events were replaced with GuiCtrl.OnNotify() and GuiCtrl.OnCommand(), which can be used with any control.

Link's Click event passes "Ctrl, ID or Index, HREF" instead of "Ctrl, Index, HREF or ID", and does not automatically execute HREF if a Click callback is registered.

ListView's Click, DoubleClick and ContextMenu (when triggered by a right-click) events now report the item which was clicked (or 0 if none) instead of the focused item.

ListView's I event was split into multiple named events, except for the f (de-focus) event, which was excluded because it is implied by F (ItemFocus).

ListView's e (ItemEdit) event is ignored if the user cancels.

Slider's Change event is raised more consistently than the v1 g-label; i.e. it no longer ignores changes made by the mouse wheel by default. See Detecting Changes (Slider) for details.

The BS_NOTIFY style is now added automatically as needed for Button, CheckBox and Radio controls. It is no longer applied by default to Radio controls.

Focus (formerly F) and LoseFocus (formerly f) are supported by more (but not all) control types.

Setting an Edit control's text with Edit.Value or Edit.Text does not trigger the control's Change event, whereas GuiControl would trigger the control's g-label.

LV/TV.Add/Modify now suppress item-change events, so such events should only be raised by user action or SendMessage.

Removed

+Delimiter

+HwndOutputVar (use GuiObj.Hwnd or GuiCtrl.Hwnd instead)

+Label

+LastFoundExist

Gui GuiName: Default

Control Options

+/-Background is interpreted and supported more consistently. All controls which supported "Gui Color" now support +BackgroundColor and +BackgroundDefault (synonymous with -Background), not just ListView/TreeView/StatusBar/Progress.

GuiObj.Add defaults to y+m/x+m instead of yp/xp when xp/yp or xp+0/yp+0 is used. In other words, the control is placed below/to the right of the previous control instead of at exactly the same

position. If a non-zero offset is used, the behaviour is the same as in v1. To use exactly the same position, specify xp yp together.

x+m and y+m can be followed by an additional offset, such as x+m+10 (x+m10 is also valid, but less readable).

Choose no longer serves as a redundant (undocumented) way to specify the value for a MonthCal. Just use the Text parameter, as before.

GuiControlGet

Empty sub-command

GuiControlGet's empty sub-command had two modes: the default mode, and text mode, where the fourth parameter was the word Text. If a control type had no single "value", GuiControlGet defaulted to returning the result of GetWindowText (which isn't always visible text). Some controls had no visible text, or did not support retrieving it, so completely ignored the fourth parameter. By contrast, GuiCtrl.Text returns display text, hidden text (the same text returned by ControlGetText) or nothing at all.

The table below shows the closest equivalent property or function for each mode of GuiControlGet and control type.

Control Default Text Notes

ActiveX .Value .Text Text is hidden. See below.

Button .Text

CheckBox .Value .Text

ComboBox .Text ControlGetText() Use Value instead of Text if AltSubmit was used (but Value returns 0 if Text does not match a list item). Text performs case-correction, whereas ControlGetText returns the Edit field's content.

Custom .Text

DateTime .Value

DDL .Text Use Value instead of Text if AltSubmit was used.

Edit .Value

GroupBox .Text

Hotkey .Value

Link .Text

ListBox.Text ControlGetText() Use Value instead of Text if AltSubmit was used. Text returns the selected item's text, whereas ControlGetText returns hidden text. See below.

ListView .Text Text is hidden.

MonthCal .Value

Picture .Value

Progress .Value

Radio .Value .Text

Slider .Value

StatusBar .Text

Tab .Text ControlGetText() Use Value instead of Text if AltSubmit was used. Text returns the selected tab's text, whereas ControlGetText returns hidden text.

Text .Text

TreeView .Text Text is hidden.

UpDown .Value

ListBox: For multi-select ListBox, Text and Value return an array instead of a pipe-delimited list.

ActiveX: GuiCtrl.Value returns the same object each time, whereas GuiControlGet created a new wrapper object each time. Consequently, it is no longer necessary to retain a reference to an ActiveX object for the purpose of keeping a ComObjConnect connection alive.

Other sub-commands

Pos → GuiCtrl.GetPos()

Focus → GuiObj.FocusedCtrl; returns a GuiControl object instead of the ClassNN.

FocusV → GuiObj.FocusedCtrl.Name

Hwnd → GuiCtrl.Hwnd; returns a pure integer, not a hexadecimal string.

Enabled/Visible/Name → GuiCtrl properties of the same name.

GuiControl

(Blank) and Text sub-commands

The table below shows the closest equivalent property or function for each mode of GuiControl and control type.

Control	(Blank)	Text	Notes
ActiveX	N/A		Command had no effect.
Button		.Text	
CheckBox	.Value	.Text	
ComboBox	.Delete/Add/Choose	.Text	
Custom		.Text	
DateTime	.Value	.SetFormat()	
DDL	.Delete/Add/Choose		
Edit	.Value		
GroupBox		.Text	
Hotkey	.Value		
Link		.Text	
ListBox	.Delete/Add/Choose		
ListView	N/A		Command had no effect.

MonthCal .Value

Picture .Value

Progress .Value Use the += operator instead of the + prefix.

Radio .Value .Text

Slider .Value Use the += operator instead of the + prefix.

StatusBar .Text or SB.SetText()

Tab .Delete/Add/Choose

Text .Text

TreeView N/A Command had no effect.

UpDown .Value Use the += operator instead of the + prefix.

Other sub-commands

Move → GuiCtrl.Move(x, y, w, h)

MoveDraw → GuiCtrl.Move(x, y, w, h), GuiCtrl.Redraw()

Focus → GuiCtrl.Focus(), which now uses WM_NEXTDLGCTL instead of SetFocus, so that focusing a Button temporarily sets it as the default, consistent with tabbing to the control.

Enable/Disable → set GuiCtrl.Enabled

Hide/Show → set GuiCtrl.Visible

Choose → GuiCtrl.Choose(n), where in is a pure integer. The |n or ||n mode is not supported (use ControlChoose instead, if needed).

ChooseString → GuiCtrl.Choose(s), where s is not a pure integer. The |n or ||n mode is not supported. If the string matches multiple items in a multi-select ListBox, Choose() selects them all, not just the first.

Font → GuiCtrl.SetFont()

+/-Option → GuiCtrl.Opt("+/-Option")

Other Changes

Progress Gui controls no longer have the PBS_SMOOTH style by default, so they are now styled according to the system visual style.

The default margins and control sizes (particularly for Button controls) may differ slightly from v1 when DPI is greater than 100%.

Picture controls no longer delete their current image when they fail to set a new image via GuiCtrl.Value := "new image.png". However, removing the current image with GuiCtrl.Value := "" is permitted.

Error Handling

OnError is now called for critical errors prior to exiting the script. Although the script might not be in a state safe for execution, the attempt is made, consistent with OnExit.

Runtime errors no longer set Exception.What to the currently running user-defined function or sub (but this is still done when calling Error() without the second parameter). This gives What a clearer purpose: a function name indicates a failure of that function (not a failure to call the function or evaluate its parameters). What is blank for expression evaluation and control flow errors (some others may also be blank).

Exception objects thrown by runtime errors can now be identified as instances of the new Error class or a more specific subclass. Error objects have a Stack property containing a stack trace. If the What parameter specifies the name of a running function, File and Line are now set based on which line called that function.

Try-catch syntax has changed to allow the script to catch specific error classes, while leaving others uncaught. See Catch below for details.

Continuable Errors

In most cases, error dialogs now provide the option to continue the current thread (vs. exiting the thread). COM errors now exit the thread when choosing not to continue (vs. exiting the entire script).

Scripts should not rely on this: If the error was raised by a built-in function, continuing causes it to return "". If the error was raised by the expression evaluator (such as for an invalid dynamic reference or divide by zero), the expression is aborted and yields "" (if used as a control flow statement's parameter).

In some cases the code does not support continuation, and the option to continue should not be shown. The option is also not shown for critical errors, which are designed to terminate the script.

OnError callbacks now take a second parameter, containing one of the following values:

"Return": Returning -1 will continue the thread, while 0 and 1 act as before.

"Exit": Continuation not supported. Returning non-zero stops further processing but still exits the thread.

"ExitApp": This is a critical error. Returning non-zero stops further processing but the script is still terminated.

ErrorLevel

ErrorLevel has been removed. Scripts are often (perhaps usually) written without error-checking, so the policy of setting ErrorLevel for errors often let them go undetected. An immediate error message may seem a bit confrontational, but is generally more helpful.

Where ErrorLevel was previously set to indicate an error condition, an exception is thrown instead, with a (usually) more helpful error message.

Commands such as "Process Exist" which used it to return a value now simply return that value (e.g. pid := ProcessExist()) or something more useful (e.g. hwnd := GroupActivate(group)).

In some cases ErrorLevel was used for a secondary return value.

Sort with the U option no longer returns the number of duplicates removed.

Input was removed. It was superseded by InputHook. A few lines of code can make a simple replacement which returns an InputHook object containing the results instead of using ErrorLevel and an OutputVar.

InputBox returns an object with result (OK, Cancel or Timeout) and value properties.

File functions which previously stored the number of failures in ErrorLevel now throw it in the Extra property of the thrown exception object.

SendMessage timeout is usually an anomolous condition, so causes a TimeoutError to be thrown. TargetError and OSError may be thrown under other conditions.

The UseErrorLevel modes of the Run and Hotkey functions were removed. This mode predates the addition of Try/Catch to the language. Menu and Gui had this mode as well but were replaced with objects (which do not use ErrorLevel).

Expressions

A load-time error is raised for more syntax errors than in v1, such as:

Empty parentheses (except adjoining a function name); e.g. x ()

Prefix operator used on the wrong side or lacking an operand; e.g. x!

Binary operator with less than two operands.

Ternary operator with less than three operands.

Target of assignment not a writable variable or property.

An exception is thrown when any of the following failures occur (instead of ignoring the failure or producing an empty string):

Attempting math on a non-numeric value. (Numeric strings are okay.)

Divide by zero or other invalid/unsupported input, such as (-1)**1.5. Note that some cases are newly detected as invalid, such as 0**0 and a<<b or a>>b where b is not in the range 0..63.

Failure to allocate memory for a built-in function's return value, concatenation or the expression's result.

Stack underflow (typically caused by a syntax error).

Attempted assignment to something which isn't a variable (or array element).

Attempted assignment to a read-only variable.

Attempted double-deref with an empty name, such as fn(%empty%).

Failure to execute a dynamic function call or method call.

A method/property invocation fails because the value does not implement that method/property. (For associative arrays in v1, only a method call can cause this.)

An object-assignment fails due to memory allocation failure.

Some of the conditions above are detected in v1, but not mid-expression; for instance, A_AhkPath := x is detected in v1 but y := x, A_AhkPath := x is only detected in v2.

Standalone use of the operators +=, -=, -- and ++ no longer treats an empty variable as 0. This differs from v1, where they treated an empty variable as 0 when used standalone, but not mid-expression or with multi-statement comma.

Functions

Functions generally throw an exception on failure. In particular:

Errors due to incorrect use of DllCall, RegExMatch and RegExReplace were fairly common due to their complexity, and (like many errors) are easier to detect and debug if an error message is shown immediately.

Math functions throw an exception if any of their inputs are non-numeric, or if the operation is invalid (such as division by zero).

Functions with a WinTitle parameter (with exceptions, such as WinClose's ahk_group mode) throw if the target window or control is not found.

Exceptions are thrown for some errors that weren't previously detected, and some conditions that were incorrectly marked as errors (previously by setting ErrorLevel) were fixed.

Some error messages have been changed.

Catch

The syntax for Catch has been changed to provide a way to catch specific error classes, while leaving others uncaught (to transfer control to another Catch further up the call stack, or report the error and exit the thread). Previously this required catching thrown values of all types, then checking type and re-throwing. For example:

```
; Old (uses obsolete v2.0-a rules for demonstration since v1 had no `is` or Error classes)
try
   SendMessage msg,,, "Control1", "The Window"
catch err
   if err is TimeoutError
      MsgBox "The Window is unresponsive"
   else
      throw err

; New
try
```

```
    SendMessage msg,,, "Control1", "The Window"
catch TimeoutError
    MsgBox "The Window is unresponsive"
```

Variations:

catch catches an Error instance.

catch as err catches an Error instance, which is assigned to err.

catch ValueError as err catches a ValueError instance, which is assigned to err.

catch ValueError, TypeError catches either type.

catch ValueError, TypeError as err catches either type and assigns the instance to err.

catch Any catches anything.

catch (MyError as err) permits parentheses, like most other control flow statements.

If try is used without finally or catch, it acts as though it has a catch with an empty block. Although that sounds like v1, now catch on its own only catches instances of Error. In most cases, try on its own is meant to suppress an Error, so no change needs to be made. However, the direct equivalent of v1 try something() in v2 is:

```
try something()
catch Any
{}
```

Prioritising the error type over the output variable name might encourage better code; handling the expected error as intended without suppressing or mishandling unexpected errors that should have been reported.

As values of all types can be thrown, any class is valid for the filter (e.g. String or Map). However, the class prototypes are resolved at load time, and must be specified as a full class name and not an arbitrary expression (similar to y in class x extends y).

While a catch statement is executing, throw can be used without parameters to re-throw the exception (avoiding the need to specify an output variable just for that purpose). This is supported even within a nested try...finally, but not within a nested try...catch. The throw does not need to be physically contained by the catch statement's body; it can be used by a called function.

An else can be present after the last catch; this is executed if no exception is thrown within try.

Keyboard, Mouse, Hotkeys and Hotstrings

Fewer VK to SC and SC to VK mappings are hard-coded, in theory improving compatibility with non-conventional custom keyboard layouts.

The key names "Return" and "Break" were removed. Use "Enter" and "Pause" instead.

The presence of AltGr on each keyboard layout is now always detected by reading the KLLF_ALTGR flag from the keyboard layout DLL. (v1.1.28+ Unicode builds already use this method.) The fallback methods of detecting AltGr via the keyboard hook have been removed.

Mouse wheel hotkeys set A_EventInfo to the wheel delta as reported by the mouse driver instead of dividing by 120. Generally it is a multiple of 120, but some mouse hardware/drivers may report wheel movement at a higher resolution.

Hotstrings now treat Shift+Backspace the same as Backspace, instead of transcribing it to `b within the hotstring buffer.

Hotstrings use the first pair of colons (::) as a delimiter rather than the last when multiple pairs of colons are present. In other words, colons (when adjacent to another colon) must be escaped in the trigger text in v2, whereas in v1 they must be escaped in the replacement. Note that with an odd number of consecutive colons, the previous behaviour did not consider the final colon as part of a pair. For example, there is no change in behaviour for ::1:::2 (1 → :2), but ::3::::4 is now 3 → ::4 rather than 3:: → 4.

Hotstrings no longer escape colons in pairs, which means it is now possible to escape a single colon at the end of the hotstring trigger. For example, ::5`:::6 is now 5: → 6 rather than an error, and ::7`::::8 is now 7: → :8 rather than 7:: → 8. It is best to escape every literal colon in these cases to avoid confusion (but a single isolated colon need not be escaped).

Hotstrings with continuation sections now default to Text mode instead of Raw mode.

Hotkeys now mask the Win/Alt key on release only if it is logically down and the hotkey requires the Win/Alt key (with #/! or a custom prefix). That is, hotkeys which do not require the Win/Alt key no longer mask Win/Alt-up when the Win/Alt key is physically down. This allows hotkeys which send {Blind}{LWin up} to activate the Start menu (which was already possible if using a remapped key such as AppsKey::RWin).

Other

Windows 2000 and Windows XP support has been dropped.

AutoHotkey no longer overrides the system ForegroundLockTimeout setting at startup.

This was done by calling SystemParametersInfo with the SPI_SETFOREGROUNDLOCKTIMEOUT action, which affects all applications for the current user session. It does not persist after logout, but was still undesirable to some users.

User bug reports (and simple logic) indicate that if it works, it allows the focus to be stolen by programs which aren't specifically designed to do so.

Some testing on Windows 10 indicated that it had no effect on anything; calls to SetForegroundWindow always failed, and other workarounds employed by WinActivate were needed and effective regardless of timeout. SPI_GETFOREGROUNDLOCKTIMEOUT was used from a separate process to verify that the change took effect (it sometimes doesn't).

It can be replicated in script easily:

DllCall("SystemParametersInfo", "int", 0x2001, "int", 0, "ptr", 0, "int", 2)

RegEx newline matching defaults to (*ANYCRLF) and (*BSR_ANYCRLF); `r and `n are recognized in addition to `r`n. The `a option implicitly enables (*BSR_UNICODE).

RegEx callout functions can now be variadic. Callouts specified via a pcre_callout variable can be any callable object, or pcre_callout itself can be directly defined as a function (perhaps a nested function). As the function and variable namespaces were merged, a callout pattern such as (?C:fn) can also refer to a local or global variable containing a function object, not just a user-defined function.

Scripts read from stdin (e.g. with AutoHotkey.exe *) no longer include the initial working directory in A_ScriptFullPath or the main window's title, but it is used as A_ScriptDir and to locate the local Lib folder.

Settings changed by the auto-execute thread now become the default settings immediately (for threads launched after that point), rather than after 100ms and then again when the auto-execute thread finishes.

The following limits have been removed by utilizing dynamic allocations:

Maximum line or continuation section length of 16,383 characters.

Maximum 512 tokens per expression (MAX_TOKENS).

Arrays internal to the expression evaluator which were sized based on MAX_TOKENS are now based on precalculated estimates of the required sizes, so performance should be similar but stack usage is somewhat lower in most cases. This might increase the maximum recursion depth of user-defined functions.

Maximum 512 var or function references per arg (but MAX_TOKENS was more limiting for expressions anyway).

Maximum 255 specified parameter values per function call (but MAX_TOKENS was more limiting anyway).

ListVars now shows static variables separately to local variables. Global variables declared within the function are also listed as static variables (this is a side-effect of new implementation details, but is kept as it might be useful in scripts with many global variables).

The (undocumented?) "lazy var" optimization was removed to reduce code size and maintenance costs. This optimization improved performance of scripts with more than 100,000 variables.

Tray menu: The word "This" was removed from "Reload This Script" and "Edit This Script", for consistency with "Pause Script" and the main window's menu options.

YYYYMMDDHH24MISS timestamp values are now considered invalid if their length is not an even number between 4 and 14 (inclusive).

Persistence

Scripts are "persistent" while at least one of the following conditions is satisfied:

At least one hotkey or hotstring has been defined by the script.

At least one Gui (or the script's main window) is visible.

At least one script timer is currently enabled.

At least one OnClipboardChange callback function has been set.

At least one InputHook is active.

Persistent() or Persistent(true) was called and not reversed by calling Persistent(false).

If one of the following occurs and none of the above conditions are satisfied, the script terminates.

The last script thread finishes.

A Gui is closed or destroyed.

The script's main window is closed (but destroying it causes the script to exit regardless of persistence, as before).

An InputHook with no OnEnd callback ends.

For flexibility, OnMessage does not make the script automatically persistent.

By contrast, v1 scripts are "persistent" when at least one of the following is true:

At least one hotkey or hotstring has been defined by the script.

Gui or OnMessage() appears anywhere in the script.

The keyboard hook or mouse hook is installed.

Input has been called.

#Persistent was used.

Threads

Threads start out with an uninterruptible timeout of 17ms instead of 15ms. 15 was too low since the system tick count updates in steps of 15 or 16 minimum; i.e. if the tick count updated at exactly the wrong moment, the thread could become interruptible even though virtually no time had passed.

Threads which start out uninterruptible now remain so until at least one line has executed, even if the uninterruptible timeout expires first (such as if the system suspends the process immediately after the thread starts in order to give CPU time to another process).

#MaxThreads and #MaxThreadsPerHotkey no longer make exceptions for any subroutine whose first line is one of the following functions: ExitApp, Pause, Edit, Reload, KeyHistory, ListLines, ListVars, or ListHotkeys.

Default Settings

#NoEnv is the default behaviour, so the directive itself has been removed. Use EnvGet instead if an equivalent built-in variable is not available.

SendMode defaults to Input instead of Event.

Title matching mode defaults to 2 instead of 1.

SetBatchLines has been removed, so all scripts run at full speed (equivalent to SetBatchLines -1 in v1).

The working directory defaults to A_ScriptDir. A_InitialWorkingDir contains the working directory which was set by the process which launched AutoHotkey.

#SingleInstance prompt behaviour is default for all scripts; #SingleInstance on its own activates Force mode. #SingleInstance Prompt can also be used explicitly, for clarity or to override a previous directive.

CoordMode defaults to Client (added in v1.1.05) instead of Window.

The default codepage for script files (but not files read by the script) is now UTF-8 instead of ANSI (CP0). This can be overridden with the /CP command line switch, as before.

#MaxMem was removed, and no artificial limit is placed on variable capacity.

Command Line

Command-line args are no longer stored in a pseudo-array of numbered global vars; the global variable A_Args (added in v1.1.27) should be used instead.

The /R and /F switches were removed. Use /restart and /force instead.

/validate should be used in place of /iLib when AutoHotkey.exe is being used to check a script for syntax errors, as the function library auto-include mechanism was removed.

/ErrorStdOut is now treated as one of the script's parameters, not built-in, in either of the following cases:

When the script is compiled, unless /script is used.

When it has a suffix not beginning with = (where previously the suffix was ignored).