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 Values formerly returned via ErrorLevel are returned by other means, replaced with exceptions, superseded or simply not returned.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 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 ¶m 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 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). Copyright © 2003-2023 www.autohotkey.com - LIC: GNU GPLv2Alphabetical Function Index Click on a function name for details. Entries in large font are the most commonly used. Go to entries starting with: E, I, M, S, W, # Name Description { ... } (Block) Blocks are one or more statements enclosed in braces. Typically used with function definitions and control flow statements. { ... } / Object Creates an Object from a list of property name and value pairs. [...] / Array Creates an Array from a sequence of parameter values. Abs Returns the absolute value of the specified number. ASin Returns the arcsine (the number whose sine is the specified number) in radians. ACos Returns the arccosine (the number whose cosine is the specified number) in radians. ATan Returns the arctangent (the number whose tangent is the specified number) in radians. BlockInput Disables or enables the user's ability to interact with the computer via keyboard and mouse. Break Exits (terminates) any type of loop statement. Buffer Creates a Buffer, which encapsulates a block of memory for use with other functions. CallbackCreate Creates a machine-code address that when called, redirects the call to a function in the script. CallbackFree Frees a callback created by CallbackCreate. CaretGetPos Retrieves the current position of the caret (text insertion point). Catch Specifies the code to execute if a value or error is thrown during execution of a Try statement. Ceil Returns the specified number rounded up to the nearest integer (without any .00 suffix). Chr Returns the string (usually a single character) corresponding to the character code indicated by the specified number. Click Clicks a mouse button at the specified coordinates. It can also hold down a mouse button, turn the mouse wheel, or move the mouse. ClipboardAll Creates an object containing everything on the clipboard (such as pictures and formatting). ClipWait Waits until the clipboard contains data. ComCall Calls a native COM interface method by index. ComObjActive Retrieves a registered COM object. ComObjArray Creates a SafeArray for use with COM. ComObjConnect Connects a COM object's event source to the script, enabling events to be handled. ComObject Creates a COM object. ComObjFlags Retrieves or changes flags which control a COM wrapper object's behaviour. ComObjFromPtr Wraps a raw IDispatch pointer (COM object) for use by the script. ComObjGet Returns a reference to an object provided by a COM component. ComObjQuery Queries a COM object for an interface or service. ComObjType Retrieves type information from a COM object. ComObjValue Retrieves the value or pointer stored in a COM wrapper object. ComValue Wraps a value, SafeArray or COM object for use by the script or for passing to a COM method. Continue Skips the rest of a loop statement's current iteration and begins a new one. ControlAddItem Adds the specified string as a new entry at the bottom of a ListBox or ComboBox. ControlChooseIndex Sets the selection in a ListBox, ComboBox or Tab control to be the specified entry or tab number. ControlChooseString Sets the selection in a ListBox or ComboBox to be the first entry whose leading part matches the specified string. ControlClick Sends a mouse button or mouse wheel event to a control. ControlDeleteItem Deletes the specified entry number from a ListBox or ComboBox. ControlFindItem Returns the entry number of a ListBox or ComboBox that is a complete match for the specified string. ControlFocus Sets input focus to a given control on a window. ControlGetChecked Returns a non-zero value if the checkbox or radio button is checked. ControlGetChoice Returns the name of the currently selected entry in a ListBox or ComboBox. ControlGetClassNN Returns the ClassNN (class name and sequence number) of the specified control. ControlGetEnabled Returns a non-zero value if the specified control is enabled. ControlGetFocus Retrieves which control of the target window has keyboard focus, if any. ControlGetHwnd Returns the unique ID number of the specified control. ControlGetIndex Returns the index of the currently selected entry or tab in a ListBox, ComboBox or Tab control. ControlGetItems Returns an array of items/rows from a ListBox, ComboBox, or DropDownList. ControlGetPos Retrieves the position and size of a control. ControlGetStyle ControlGetExStyle Returns an integer representing the style or extended style of the specified control. ControlGetText Retrieves text from a control. ControlGetVisible Returns a non-zero value if the specified control is visible. ControlHide Hides the specified control. ControlHideDropDown Hides the drop-down list of a ComboBox control. ControlMove Moves or resizes a control. ControlSend ControlSendText Sends simulated keystrokes or text to a window or control. ControlSetChecked Turns on (checks) or turns off (unchecks) a checkbox or radio button. ControlSetEnabled Enables or disables the specified control. ControlSetStyle ControlSetExStyle Changes the style or extended style of the specified control, respectively. ControlSetText Changes the text of a control. ControlShow Shows the specified control if it was previously hidden. ControlShowDropDown Shows the drop-down list of a ComboBox control. CoordMode Sets coordinate mode for various built-in functions to be relative to either the active window or the screen. Cos Returns the trigonometric cosine of the specified number. Critical Prevents the current thread from being interrupted by other threads, or enables it to be interrupted. DateAdd Adds or subtracts time from a date-time value. DateDiff Compares two date-time values and returns the difference. DetectHiddenText Determines whether invisible text in a window is "seen" for the purpose of finding the window. This affects built-in functions such as WinExist and WinActivate. DetectHiddenWindows Determines whether invisible windows are "seen" by the script. DirCopy Copies a folder along with all its sub-folders and files (similar to xcopy). DirCreate Creates a folder. DirDelete Deletes a folder. DirExist Checks for the existence of a folder and returns its attributes. DirMove Moves a folder along with all its sub-folders and files. It can also rename a folder. DirSelect Displays a standard dialog that allows the user to select a folder. DllCall Calls a function inside a DLL, such as a standard Windows API function. Download Downloads a file from the Internet. DriveEject Ejects the tray of the specified CD/DVD drive, or ejects a removable drive. DriveGetCapacity Returns the total capacity of the drive which contains the specified path, in megabytes. DriveGetFileSystem Returns the type of the specified drive's file system. DriveGetLabel Returns the volume label of the specified drive. DriveGetList Returns a string of letters, one character for each drive letter in the system. DriveGetSerial Returns the volume serial number of the specified drive. DriveGetSpaceFree Returns the free disk space of the drive which contains the specified path, in megabytes. DriveGetStatus Returns the status of the drive which contains the specified path. DriveGetStatusCD Returns the media status of the specified CD/DVD drive. DriveGetType Returns the type of the drive which contains the specified path.

DriveLock Prevents the eject feature of the specified drive from working. DriveRetract Retracts the tray of the specified CD/DVD drive. DriveSetLabel Changes the volume label of the specified drive. DriveUnlock Restores the eject feature of the specified drive. Edit Opens the current script for editing in the default editor. EditGetCurrentCol Returns the column number in an Edit control where the caret (text insertion point) resides. EditGetCurrentLine Returns the line number in an Edit control where the caret (text insert point) resides. EditGetLine Returns the text of the specified line in an Edit control. EditGetLineCount Returns the number of lines in an Edit control. EditGetSelectedText Returns the selected text in an Edit control. EditPaste Pastes the specified string at the caret (text insertion point) in an Edit control. Else Specifies one or more statements to execute if the associated statement's body did not execute. EnvGet Retrieves an environment variable. EnvSet Writes a value to a variable contained in the environment. Exit Exits the current thread. ExitApp Terminates the script. Exp Returns e (which is approximately 2.71828182845905) raised to the Nth power. FileAppend Writes text or binary data to the end of a file (first creating the file, if necessary). FileCopy Copies one or more files. FileCreateShortcut Creates a shortcut (.lnk) file. FileDelete Deletes one or more files. FileEncoding Sets the default encoding for FileRead, Loop Read, FileAppend, and FileOpen. FileExist Checks for the existence of a file or folder and returns its attributes. FileInstall Includes the specified file inside the compiled version of the script. FileGetAttrib Reports whether a file or folder is read-only, hidden, etc. FileGetShortcut Retrieves information about a shortcut (.lnk) file, such as its target file. FileGetSize Retrieves the size of a file. FileGetTime Retrieves the datetime stamp of a file or folder. FileGetVersion Retrieves the version of a file. FileMove Moves or renames one or more files. FileOpen Opens a file to read specific content from it and/or to write new content into it. FileRead Retrieves the contents of a file. FileRecycle Sends a file or directory to the recycle bin if possible, or permanently deletes it. FileRecycleEmpty Empties the recycle bin. FileSelect Displays a standard dialog that allows the user to open or save file(s). FileSetAttrib Changes the attributes of one or more files or folders. Wildcards are supported. FileSetTime Changes the datetime stamp of one or more files or folders. Wildcards are supported. Finally Ensures that one or more statements are always executed after a Try statement finishes. Float Converts a numeric string or integer value to a floating-point number. Floor Returns the specified number rounded down to the nearest integer (without any .00 suffix). For Repeats one or more statements once for each key-value pair in an object. Format Formats a variable number of input values according to a format string. FormatTime Transforms a YYYYMMDDHH24MISS timestamp into the specified date/time format. GetKeyName Retrieves the name/text of a key. GetKeyVK Retrieves the virtual key code of a key. GetKeySC Retrieves the scan code of a key. GetKeyState Checks if a keyboard key or mouse/controller button is down or up. Also retrieves controller status. GetMethod Retrieves the implementation function of a method. Goto Jumps to the specified label and continues execution. GroupActivate Activates the next window in a window group that was defined with GroupAdd. GroupAdd Adds a window specification to a window group, creating the group if necessary. GroupClose Closes the active window if it was just activated by GroupActivate or GroupDeactivate. It then activates the next window in the series. It can also close all windows in a group. GroupDeactivate Similar to GroupActivate except activates the next window not in the group. Gui() Creates and returns a new Gui object. This can be used to define a custom window, or graphical user interface (GUI), to display information or accept user input. GuiCtrlFromHwnd Retrieves the GuiControl object of a GUI control associated with the specified HWND. GuiFromHwnd Retrieves the Gui object of a GUI window associated with the specified HWND. HasBase Returns a non-zero number if the specified value is derived from the specified base object. HasMethod Returns a non-zero number if the specified value has a method by the specified name. HasProp Returns a non-zero number if the specified value has a property by the specified name. HotIf / HotIfWin... Specifies the criteria for subsequently created or modified hotkey variants. Hotkey Creates, modifies, enables, or disables a hotkey while the script is running. Hotstring Creates, modifies, enables, or disables a hotstring while the script is running. If Specifies one or more statements to execute if an expression evaluates to true. IL_Create IL_Add IL_Destroy The means by which icons are added to a ListView or TreeView control. ImageSearch Searches a region of the screen for an image. IniDelete Deletes a value from a standard format .ini file. IniRead Reads a value, section or list of section names from a standard format .ini file. IniWrite Writes a value or section to a standard format .ini file. InputBox Displays an input box to ask the user to enter a string. InputHook Creates an object which can be used to collect or intercept keyboard input. InstallKeybdHook Installs or uninstalls the keyboard hook. InstallMouseHook Installs or uninstalls the mouse hook. InStr Searches for a given occurrence of a string, from the left or the right. Integer Converts a numeric string or floating-point value to an integer. IsLabel Returns a non-zero number if the specified label exists in the current scope. IsObject Returns a non-zero number if the specified value is an object. IsSet / IsSetRef Returns a non-zero number if the specified variable has been assigned a value. KeyHistory Displays script info and a history of the most recent keystrokes and mouse clicks. KeyWait Waits for a key or mouse/controller button to be released or pressed down. ListHotkeys Displays the hotkeys in use by the current script, whether their subroutines are currently running, and whether or not they use the keyboard or mouse hook. ListLines Enables or disables line logging or displays the script lines most recently executed. ListVars Displays the script's variables: their names and current contents. ListViewGetContent Returns a list of items/rows from a ListView. LoadPicture Loads a picture from file and returns a bitmap or icon handle. Log Returns the logarithm (base 10) of the specified number. Ln Returns the natural logarithm (base e) of the specified number. Loop (normal) Performs one or more statements repeatedly: either the specified number of times or until Break is encountered. 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. Loop Reg Retrieves the contents of the specified registry subkey, one item at a time. Map Creates a Map from a list of key-value pairs. Max Returns the highest number from a set of numbers. MenuBar() Creates a MenuBar object, which can be used to define a GUI menu bar. Menu() Creates a Menu object, which can be used to create and display a menu. MenuFromHandle Retrieves the Menu or MenuBar object corresponding to a Win32 menu handle. MenuSelect Invokes a menu item from the menu bar of the specified window. Min Returns the lowest number from a set of numbers. Mod Modulo. Returns the remainder of the specified dividend divided by the specified divisor. MonitorGet Checks if the specified monitor exists and optionally retrieves its bounding coordinates. MonitorGetCount Returns the total number of monitors. MonitorGetName Returns the operating system's name of the specified monitor. MonitorGetPrimary Returns the number of the primary monitor. MonitorGetWorkArea Checks if the specified monitor exists and optionally retrieves the bounding coordinates of its working area. MouseClick Clicks or holds down a mouse button, or turns the mouse wheel. Note: The Click function is generally more flexible and easier to use. MouseClickDrag Clicks and holds the specified mouse button, moves the mouse to the destination coordinates, then releases the button. MouseGetPos Retrieves the current position of the mouse cursor, and optionally which window and control it is hovering over. MouseMove Moves the mouse cursor. MsgBox Displays the specified text in a small window containing one or more buttons (such as Yes and No). Number Converts a numeric string to a pure integer or floating-point number. NumGet Returns the binary number stored at the specified address+offset. NumPut Stores one or more numbers in binary format at the specified address+offset. ObjAddRef / ObjRelease Increments or decrements an object's reference count. ObjBindMethod Creates a BoundFunc object which calls a method of a given object. ObjHasOwnProp ObjOwnProps These functions are equivalent to built-in methods of the Object type. It is usually recommended to use the corresponding method instead. ObjGetBase Retrieves an object's base object. ObjGetCapacity Returns the current capacity of the object's internal array of properties. ObjOwnPropCount Returns the number of properties owned by an object. ObjSetBase Sets an object's base object. ObjSetCapacity Sets the current capacity of the object's internal array of own properties. OnClipboardChange Causes the specified function to be called automatically whenever the clipboard's content changes. OnError Causes the specified function to be called automatically when an unhandled error occurs. OnExit Causes the specified function to be called automatically when the script exits. OnMessage Causes the specified function to be called automatically whenever the script receives the specified message. Ord Returns the ordinal value (numeric character code) of the first character in the specified string. OutputDebug Sends a string to the debugger (if any) for display. Pause Pauses the script's current thread. Persistent Prevents the script from exiting automatically when its last thread completes, allowing it to stay running in an idle state. PixelGetColor Retrieves the color of the pixel at the specified x,y coordinates. PixelSearch Searches a region of the screen for a pixel of the specified color. PostMessage Places a message in the message queue of a window or control. ProcessClose Forces the first matching process to close. ProcessExist Checks if the specified process exists. ProcessGetName Returns the name of the specified process. ProcessGetParent Returns the process ID (PID) of the process which created the specified process. ProcessGetPath Returns the path of the specified process. ProcessSetPriority Changes the priority level of the first matching process. ProcessWait Waits for the specified process to exist. ProcessWaitClose Waits for all matching processes to close. Random Generates a pseudo-random number. RegExMatch Determines whether a string contains a pattern (regular expression). RegExReplace Replaces occurrences of a pattern (regular expression) inside a string. RegCreateKey Creates a registry key without writing a value. RegDelete Deletes a value from the registry. RegDeleteKey Deletes a subkey from the registry. RegRead Reads a value from the registry. RegWrite Writes a value to the registry. Reload Replaces the currently running instance of the script with a new one. Return Returns from a subroutine to which execution had previously jumped via function-call, Hotkey activation, or other means. Round Returns the specified number rounded to N decimal places. Run Runs an external program. RunAs Specifies a set of user credentials to use for all subsequent uses of Run and RunWait. RunWait Runs an external program and waits until it finishes. Send / SendText / SendInput / SendPlay / SendEvent Sends simulated keystrokes and mouse clicks to the active window. SendLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. SendMessage Sends a message to a window or control and waits for acknowledgement. SendMode Makes Send synonymous with SendEvent or SendPlay rather than the default (SendInput). Also makes Click and MouseMove/Click/Drag use the specified method. SetCapsLockState Sets the state of CapsLock. Can also force the key to stay on or off. SetControlDelay Sets the delay that will occur after each control-modifying function. SetDefaultMouseSpeed Sets the mouse speed that will be used if unspecified in Click and MouseMove/Click/Drag. SetKeyDelay Sets the delay that will occur after each keystroke sent by Send or ControlSend. SetMouseDelay Sets the delay that will occur after each mouse movement or click. SetNumLockState Sets the state of NumLock. Can also force the key to stay on or off. SetScrollLockState Sets the state of ScrollLock. Can also force the key to stay on or off. SetRegView Sets the registry view used by RegRead, RegWrite, RegDelete, RegDeleteKey and Loop Reg, allowing them in a 32-bit script to access the 64-bit registry view and vice versa. SetStoreCapsLockMode Whether to restore the state of CapsLock after a Send. SetTimer Causes a function to be called automatically and repeatedly at a specified time interval. SetTitleMatchMode Sets the matching behavior of the WinTitle parameter in built-in functions such as WinWait. SetWinDelay Sets the

delay that will occur after each windowing function, such as WinActivate. SetWorkingDir Changes the script's current working directory. Shutdown Shuts down, restarts, or logs off the system. Sin Returns the trigonometric sine of the specified number. Sleep Waits the specified amount of time before continuing. Sort Arranges a variable's contents in alphabetical, numerical, or random order (optionally removing duplicates). SoundBeep Emits a tone from the PC speaker. SoundGetInterface Retrieves a native COM interface of a sound device or component. SoundGetMute Retrieves a mute setting of a sound device. SoundGetName Retrieves the name of a sound device or component. SoundGetVolume Retrieves a volume setting of a sound device. SoundPlay Plays a sound, video, or other supported file type. SoundSetMute Changes a mute setting of a sound device. SoundSetVolume Changes a volume setting of a sound device. SplitPath Separates a file name or URL into its name, directory, extension, and drive. Sqrt Returns the square root of the specified number. StatusBarGetText Retrieves the text from a standard status bar control. StatusBarWait Waits until a window's status bar contains the specified string. StrCompare Compares two strings alphabetically. StrGet Copies a string from a memory address or buffer, optionally converting it from a given code page. String Converts a value to a string. StrLen Retrieves the count of how many characters are in a string. StrLower Converts a string to lowercase. StrPtr Returns the current memory address of a string. StrPut Copies a string to a memory address or buffer, optionally converting it to a given code page. StrReplace Replaces the specified substring with a new string. StrSplit Separates a string into an array of substrings using the specified delimiters. StrUpper Converts a string to uppercase. SubStr Retrieves one or more characters from the specified position in a string. Suspend Disables or enables all or selected hotkeys and hotstrings. Switch Executes one case from a list of mutually exclusive candidates. SysGet Retrieves dimensions of system objects, and other system properties. SysGetIPAddresses Returns an array of the system's IPv4 addresses. Tan Returns the trigonometric tangent of the specified number. Thread Sets the priority or interruptibility of threads. It can also temporarily disable all timers. Throw Signals the occurrence of an error. This signal can be caught by a Try-Catch statement. ToolTip Creates an always-on-top window anywhere on the screen. TraySetIcon Changes the script's tray icon (which is also used by GUI and dialog windows). TrayTip Creates a balloon message window near the tray icon. On Windows 10, a toast notification may be shown instead. Trim / LTrim / RTrim Trims characters from the beginning and/or end of a string. Try Guards one or more statements against runtime errors and values thrown by the Throw statement. Type Returns the class name of a value. Until Applies a condition to the continuation of a Loop or For-loop. VarSetStrCapacity Enlarges a variable's holding capacity or frees its memory. This is not normally needed, but may be used with DllCall or SendMessage or to optimize repeated concatenation. VerCompare Compares two version strings. While-loop Performs one or more statements repeatedly until the specified expression evaluates to false. WinActivate Activates the specified window. WinActivateBottom Same as WinActivate except that it activates the bottommost matching window rather than the topmost. WinActive Checks if the specified window is active and returns its unique ID (HWND). WinClose Closes the specified window. WinExist Checks if the specified window exists and returns the unique ID (HWND) of the first matching window. WinGetClass Retrieves the specified window's class name. WinGetClientPos Retrieves the position and size of the specified window's client area. WinGetControls Returns the control names for all controls in the specified window. WinGetControlsHwnd Returns the unique ID numbers for all controls in the specified window. WinGetCount Returns the number of existing windows that match the specified criteria. WinGetID Returns the unique ID number of the specified window. WinGetIDLast Returns the unique ID number of the last/bottommost window if there is more than one match. WinGetList Returns the unique ID numbers of all existing windows that match the specified criteria. WinGetMinMax Returns the state whether the specified window is maximized or minimized. WinGetPID Returns the Process ID number of the specified window. WinGetPos Retrieves the position and size of the specified window. WinGetProcessName Returns the name of the process that owns the specified window. WinGetProcessPath Returns the full path and name of the process that owns the specified window. WinGetStyle WinGetExStyle Returns the style or extended style (respectively) of the specified window. WinGetText Retrieves the text from the specified window. WinGetTitle Retrieves the title of the specified window. WinGetTransColor Returns the color that is marked transparent in the specified window. WinGetTransparent Returns the degree of transparency of the specified window. WinHide Hides the specified window. WinKill Forces the specified window to close. WinMaximize Enlarges the specified window to its maximum size. WinMinimize Collapses the specified window into a button on the task bar. WinMinimizeAll / WinMinimizeAllUndo Minimizes or unminimizes all windows. WinMove Changes the position and/or size of the specified window. WinMoveBottom Sends the specified window to the bottom of stack; that is, beneath all other windows. WinMoveTop Brings the specified window to the top of the stack without explicitly activating it. WinRedraw Redraws the specified window. WinRestore Unminimizes or unmaximizes the specified window if it is minimized or maximized. WinSetAlwaysOnTop Makes the specified window stay on top of all other windows (except other always-on-top windows). WinSetEnabled Enables or disables the specified window. WinSetRegion Changes the shape of the specified window to be the specified rectangle, ellipse, or polygon. WinSetStyle WinSetExStyle Changes the style or extended style of the specified window, respectively. WinSetTitle Changes the title of the specified window. WinSetTransColor Makes all pixels of the chosen color invisible inside the specified window. WinSetTransparent Makes the specified window semi-transparent. WinShow Unhides the specified window. WinWait Waits until the specified window exists. WinWaitActive / WinWaitNotActive Waits until the specified window is active or not active. WinWaitClose Waits until no matching windows can be found. #ClipboardTimeout Changes how long the script keeps trying to access the clipboard when the first attempt fails. #DllLoad Loads a DLL or EXE file before the script starts executing. #ErrorStdOut Sends any syntax error that prevents a script from launching to the standard error stream (stderr) rather than displaying a dialog. #Hotstring Changes hotstring options or ending characters. #HotIf Creates context-sensitive hotkeys and hotstrings. Such hotkeys perform a different action (or none at all) depending on any condition (an expression). #HotIfTimeout Sets the maximum time that may be spent evaluating a single #HotIf expression. #Include / #IncludeAgain Causes the script to behave as though the specified file's contents are present at this exact position. #InputLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. #MaxThreads Sets the maximum number of simultaneous threads. #MaxThreadsBuffer Causes some or all hotkeys to buffer rather than ignore keypresses when their #MaxThreadsPerHotkey limit has been reached. #MaxThreadsPerHotkey Sets the maximum number of simultaneous threads per hotkey or hotstring. #NoTrayIcon Disables the showing of a tray icon. #Requires Displays an error and quits if a version requirement is not met. #SingleInstance Determines whether a script is allowed to run again when it is already running. #SuspendExempt Exempts subsequent hotkeys and hotstrings from suspension. #UseHook Forces the use of the hook to implement all or some keyboard hotkeys. #Warn Enables or disables warnings for specific conditions which may indicate an error, such as a typo or missing "global" declaration. #WinActivateForce Skips the gentle method of activating a window and goes straight to the forceful method. Copyright © 2003-2023 www.autohotkey.com - LIC: GNU GPLv2 #NoEnv SetBatchLines, -1 SetWorkingDir % A_ScriptDir% if (A_PtrSize = 8) { try RunWait "%A_AhkPath%\..\AutoHotkeyU32.exe" "%A_ScriptFullPath%" catch MsgBox 16,, This script must be run with AutoHotkey 32-bit, due to use of the ScriptControl COM component. ExitApp } ; Change this path if the loop below doesn't find your hhc.exe, ; or leave it as-is if hhc.exe is somewhere in %PATH%. hhc := "hhc.exe" ; Try to find hhc.exe, since it's not in %PATH% by default. for i, env_var in ["ProgramFiles", "ProgramFiles(x86)", "ProgramW6432"] { EnvGet Programs, %env_var% if (Programs && FileExist(checking := Programs "\HTML Help Workshop\hhc.exe")) { hhc := checking break } } FileRead IndexJS, docs\static\source\data_index.js Overwrite("Index.hhk", INDEX_CreateHHK(IndexJS)) ; Use old sidebar: ; FileDelete, docs\static\content.js ; FileRead TocJS, docs\static\source\data_toc.js ; Overwrite("Table of Contents.hhc", TOC_CreateHHC(TocJS)) ; IniWrite, Table of Contents.hhc, Project.hhp, OPTIONS, Contents file ; IniWrite, % "AutoHotkey v2 Help,Table of Contents.hhc,Index.hhk,docs\index.htm,docs\index.htm,,,,,0x73520,,0x10200e,[200,0,1080,700],0,,,,0,,0", Project.hhp, WINDOWS, Contents ; Compile AutoHotkey.chm. RunWait %hhc% "%A_ScriptDir%\Project.hhp" Overwrite(File, Text) { FileOpen(File, "w").Write(Text) } TOC_CreateHHC(data) { ComObjError(false) sc := ComObjCreate("ScriptControl") sc.Language := "JScript" sc.ExecuteStatement(data) output = (LTrim) output .= TOC_CreateListCallback("", sc.Eval("tocData")) output .= "``n``n" return % output } TOC_CreateListCallback(byref output, data) { output .= "``n``n" Loop % data.length { i := A_Index - 1 output .= "" if data[i][0] { Transform, param_name, HTML, % data[i][0] output .= "" } if data[i][1] { Transform, param_local, HTML, % data[i][1] output .= "" } output .= "" if data[i][2] output .= TOC_CreateListCallback(output, data[i][2]) output .= "``n" } output .= "" return % output } INDEX_CreateHHK(data) { ComObjError(false) sc := ComObjCreate("ScriptControl") sc.Language := "JScript" sc.ExecuteStatement(data) data := sc.Eval("indexData") output = ( LTrim ) output .= "``n``n" Loop % data.length { i := A_Index - 1 output .= "" Transform, param_name, HTML, % data[i][0] output .= "" Transform, param_local, HTML, % data[i][1] output .= "" output .= "``n" } output .= "" output .= "``n``n``n" return % output } Redirecting to docs/. Redirecting to docs/. Redirecting to docs/. # download-Autohotkey-v2-Documentation download Autohotkey-v2 Documentation ahkv2 docs Version 2.0.2 https://www.autohotkey.com Â© 2014 Steve Gray, Chris Mallett, portions Â© AutoIt Team and various others Software License: GNU General Public License Quick Reference Getting started: How to use the program Tutorials: How to run example code How to write hotkeys How to send keystrokes How to run programs How to manage windows Beginner tutorial by tidbit Text editors with AutoHotkey support Frequently asked questions Scripts: Concepts and conventions: explanations of various things you need to know. Scripting language: how to write scripts. Miscellaneous topics List of built-in functions Variables and expressions How to use functions Objects Interactive debugging Keyboard and mouse: Hotkeys (mouse, joystick and keyboard shortcuts) Hotstrings and auto-replace Remapping keys and buttons List of keys, mouse buttons and joystick controls Other: DllCall RegEx quick reference Acknowledgements A special thanks to Jonathan Bennett, whose generosity in releasing AutoIt v2 as free software in 1999 served as an inspiration and time-saver for myself and many others worldwide. In addition, many of AutoHotkey's enhancements to the AutoIt v2 command set, as well as the Window Spy and the old script compiler, were adapted directly from the AutoIt v3 source code. So thanks to Jon and the other AutoIt authors for those as well. Finally, AutoHotkey would not be what it is today without these other individuals. ~ Chris Mallett Copyright Â© 2003-2023 - LIC: GNU GPLv2 from pathlib import Path cwd = Path.cwd() x = "" for file in cwd.rglob("*"): try: with open(file, 'r') as f: y = str(f.read()) x = str(x + y) except: continue with open("docs.html", "w") as f: f.write(x) Debugging Clients | AutoHotkey v2 Debugging Clients Additional debugging features are supported via DBGp, a common debugger protocol for languages and debugger UI communication. See Interactive Debugging for more details. Some UIs or "clients" known to be compatible with AutoHotkey are listed on this page: SciTE4AutoHotkey XDebugClient Notepad++ DBGp

Plugin Script-based Clients Command-line Client Others SciTE4AutoHotkey SciTE4AutoHotkey is a free, SciTE-based AutoHotkey script editor. In addition to DBGp support, it provides syntax highlighting, calltips/parameter info and auto-complete for AutoHotkey, and other useful editing features and scripting tools. Debugging features include: Breakpoints. Run, Step Over/Into/Out. View the call stack. List name and contents of variables in local/global scope. Hover over variable to show contents. Inspect or edit variable contents. View structure of objects. https://www.autohotkey.com/scite4ahk/ Visual Studio Code The vscode-autohotkey-debug extension enables Visual Studio Code to act as a debugger client for AutoHotkey. The extension has support for all basic debugging features as well as some more advanced features, such as breakpoint directives (as comments) and conditional breakpoints. XDebugClient XDebugClient is a simple open-source front-end DBGp client based on the .NET Framework 2.0. XDebugClient was originally designed for PHP with Xdebug, but a custom build compatible with AutoHotkey is available below. Changes: Allow the debugger engine to report a language other than "php". Added AutoHotkey syntax highlighting. Automatically listen for a connection from the debugger engine, rather than waiting for the user to click Start Listening. Truncate property values at the first null-character, since AutoHotkey currently returns the entire variable contents and XDebugClient has no suitable interface for displaying binary content. Download: Binary; Source Code (also see SharpDevelop, Dockpanel Suite and Advanced TreeView.) Usage: Launch XDebugClient. Launch AutoHotkey /Debug. XDebugClient should automatically open the script file. Click the left margin to set at least one breakpoint. Choose Run from the Debug menu, or press F5. When execution hits a breakpoint, use the Debug menu or shortcut keys to step through or resume the script. Features: Syntax highlighted, read-only view of the source code. Breakpoints. Run, Step Over/Into/Out. View the call stack. Inspect variables - select a variable name, right-click, Inspect. Issues: The user interface does not respond to user input while the script is running. No mechanisms are provided to list variables or set their values. Notepad++ DBGp Plugin A DBGp client is available as a plugin for Notepad++ 32-bit. It is designed for PHP, but also works with AutoHotkey. The plugin has not been updated since 2012, and is not available for Notepad++ 64-bit. Download: See DBGp plugin for Notepad++. Usage: Launch Notepad++. Configure the DBGp plugin via Plugins, DBGp, Config... Note: File Mapping must be configured. Most users will not be debugging remotely, and therefore may simply put a checkmark next to Bypass all mapping (local windows setup). Show the debugger pane via the toolbar or Plugins, DBGp, Debugger. Open the script file to be debugged. Set at least one breakpoint. Launch AutoHotkey /Debug. Use the debugger toolbar or shortcut keys to control the debugger. Features: Syntax highlighting, if configured by the user. Breakpoints. Run, Step Over/Into/Out, Run to cursor, Stop. View local/global variables. Watch user-specified variables. View the call stack. Hover over a variable to view its contents. User-configurable shortcut keys - Settings, Shortcut Mapper..., Plugin commands. Issues: Hovering over a single-letter variable name does not work - for instance, hovering over a will attempt to retrieve a or a . Hovering over text will attempt to retrieve a variable even if the text contains invalid characters. Notepad++ becomes unstable if property_get fails, which is particularly problematic in light of the above. As a workaround, AutoHotkey sends an empty property instead of an error code when a non-existent or invalid variable is requested. Script-based Clients A script-based DBGp library and example clients are available from GitHub. dbgp_console.ahk: Simple command-line client. dbgp_test.ahk: Demonstrates asynchronous debugging. dbgp_listvars.ahk: Example client which justs lists the variables of all running scripts. GitHub: Lexikos / dbgp The DebugVars script provides a graphical user interface for inspecting and changing the contents of variables and objects in any running script (except compiled scripts). It also serves as a demonstration of the dbgp.ahk library. GitHub: Lexikos / DebugVars Command-line Client A command-line client is available from xdebug.org, however this is not suitable for most users as it requires a decent understanding of DBGp (the protocol). Others A number of other DBGp clients are available, but have not been tested with AutoHotkey. For a list, see Xdebug: Documentation. Changes & New Features | AutoHotkey v2 Changes & New Features Changes from v1.1 to v2.0 covers the differences between v1.1 and v2.0. For full technical details of changes, refer to GitHub. 2.0.2 - January 2, 2023 Fixed Short DllCall arg type and undefined behaviour for invalid types. Fixed (non-string) file version number for AutoHotkey binaries. Fixed parameter type errors to show the correct parameter number. 2.0.1 - January 1, 2023 Fixed Func.IsOptional(1) returning 0 in some cases where it shouldn't. Fixed Gui event handler functions to not drop the Gui parameter when the Gui is its own event sink. Fixed COM errors to not show "(null)" when no description is available. Fixed ToolTips intermittently appearing at the wrong position. Fixed __Enum(unset) to permit a second variable for Array, Match and Gui. Fixed #include <> error messages to show "Script library" rather than "Function library". Fixed new threads being unable to prevent a message check with Critical. Optimized conversion of DllCall type names. Made some trivial but effective code size optimizations. Pre-Release For a history of changes prior to the v2.0.0 release, refer to the following (but note some changes were superseded): v2.0 release candidates v2.0-beta releases v2.0-alpha releases Binary Compatibility | AutoHotkey v2 Binary Compatibility This document contains some topics which are sometimes important when dealing with external libraries or sending messages to a control or window. Unicode vs ANSI Buffer DllCall NumPut / NumGet Pointer Size Unicode vs ANSI Note: This section builds on topics covered in other parts of the documentation: Strings, String Encoding. Within a string (text value), the numeric character code and size (in bytes) of each character depends on the encoding of the string. These details are typically important for scripts which do any of the following: Pass strings to external functions via DllCall. Pass strings via PostMessage or SendMessage. Manipulate strings directly via NumPut/NumGet. Allocate a Buffer to hold a specific number of characters. AutoHotkey v2 natively uses Unicode (UTF-16), but some external libraries or window messages might require ANSI strings. ANSI: Each character is one byte (8 bits). Character codes above 127 depend on your system's language settings (or the codepage chosen when the text was encoded, such as when it is written to a file). Unicode: Each character is two bytes (16 bits). Character codes are as defined by the UTF-16 format. Semantic note: Technically, some Unicode characters are represented by two 16-bit code units, collectively known as a "surrogate pair." Similarly, some ANSI code pages (commonly known as Double Byte Character Sets) contain some double-byte characters. However, for practical reasons these are almost always treated as two individual units (referred to as "characters" for simplicity). Buffer When allocating a Buffer, take care to calculate the correct number of bytes for whichever encoding is required. For example: ansi_buf := Buffer (capacity_in_chars) utf16_buf := Buffer(capacity_in_chars * 2) If an ANSI or UTF-8 string will be written into the buffer with StrPut, do not use StrLen to determine the buffer size, as the ANSI or UTF-8 length may differ from the native (UTF-16) length. Instead, use StrPut to calculate the required buffer size. For example: required_bytes := StrPut(source_string, "cp0") ansi_buf := Buffer(required_bytes) StrPut(source_string, ansi_buf) DllCall When the "Str" type is used, it means a string in the native format of the current build. Since some functions may require or return strings in a particular format, the following string types are available: Char SizeC / Win32 TypesEncoding WStr16-bitwchar_t*, WCHAR*, LPWSTR, LPCWSTRUTF-16 AStr8-bitchar*, CHAR*, LPSTR, LPCSTRANSI (the system default ANSI code page) Str--TCHAR*, LPTSTR, LPCTSTREquivalent to WStr in AutoHotkey v2. If "Str" or "WStr" is used for a parameter, the address of the string is passed to the function. For "AStr", a temporary ANSI copy of the string is created and its address is passed instead. As a general rule, "AStr" should not be used for an output parameter since the buffer is only large enough to hold the input string. Note: "AStr" and "WStr" are equally valid for parameters and the function's return value. In general, if a script calls a function via DllCall which accepts a string as a parameter, one or more of the following approaches must be taken: If both Unicode (W) and ANSI (A) versions of the function are available, omit the W or A suffix and use the "Str" type for input parameters or the return value. For example, the DeleteFile function is exported from kernel32.dll as DeleteFileA and DeleteFileW. Since DeleteFile itself doesn't really exist, DllCall automatically tries DeleteFileW: DllCall("DeleteFile", "Ptr", StrPtr(filename)) DllCall("DeleteFile", "Str", filename) In both cases, the address of the original unmodified string is passed to the function. In some cases this approach may backfire, as DllCall adds the W suffix only if no function could be found with the original name. For example, shell32.dll exports ExtractIconExW, ExtractIconExA and ExtractIconEx with no suffix, with the last two being equivalent. In that case, omitting the W suffix causes the ANSI version to be called. If the function accepts a specific type of string as input, the script may use the appropriate string type: DllCall("DeleteFileA", "AStr", filename) DllCall("DeleteFileW", "WStr", filename) If the function has a string parameter used for output, the script must allocate a buffer as described above and pass it to the function. If the parameter accepts input, the script must also convert the input string to the appropriate format; StrPut can be used for this. NumPut / NumGet When NumPut or NumGet are used with strings, the offset and type must be correct for the given type of string. The following may be used as a guide: ; 8-bit/ANSI strings: size_of_char=1 type_of_char="UChar" ; 16-bit/UTF-16 strings: size_of_char=2 type_of_char="UShort" nth_char := NumGet(buffer_or_address, (n-1)*size_of_char, type_of_char) NumPut(type_of_char, nth_char, buffer_or_address, (n-1) *size_of_char) For the first character, n should have the value 1. Pointer Size Pointers are 4 bytes in 32-bit builds and 8 bytes in 64-bit builds. Scripts using structures or DllCalls may need to account for this to run correctly on both platforms. Specific areas which are affected include: Offset calculation for fields in structures which contain one or more pointers. Size calculation for structures containing one or more pointers. Type names used with DllCall, NumPut or NumGet. For size and offset calculations, use A_PtrSize. For DllCall, NumPut and NumGet, use the Ptr type where appropriate. Remember that the offset of a field is usually the total size of all fields preceding it. Also note that handles (including types like HWND and HBITMAP) are essentially pointer-types. /* typedef struct _PROCESS_INFORMATION { HANDLE hProcess; // Ptr HANDLE hThread; DWORD dwProcessId; // UInt (4 bytes) DWORD dwThreadId; } PROCESS_INFORMATION, *LPPROCESS_INFORMATION; */ pi := Buffer(A_PtrSize*2 + 8) ; Ptr + Ptr + UInt + UInt DllCall("CreateProcess", , "Ptr", &pi,) hProcess := NumGet(pi, 0) ; Defaults to "Ptr". hThread := NumGet(pi, A_PtrSize) ; dwProcessId := NumGet(pi, A_PtrSize*2, "UInt") dwProcessId := NumGet(pi, A_PtrSize*2 + 4, "UInt") Concepts and Conventions | AutoHotkey v2 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 Values Strings Numbers Boolean Nothing Objects Object Protocol Variables Uninitialized Variables Built-in Variables Environment Variables Variable References (VarRef) Caching Functions Methods Control Flow Details 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: Abc 6598990 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. Frequently Asked Questions (FAQ) | AutoHotkey v2 Frequently Asked Questions (FAQ) Table of Contents General Troubleshooting What can I do if AutoHotkey won't install? How do I restore the right-click context menu options for .ahk files? Why doesn't my script work on Windows xxx even though it worked on a previous version? How do I work around problems caused by User Account Control (UAC)? I can't edit my script via tray icon because it won't start due to an error. What should I do? How can I find and fix errors in my code? Why is the Run function unable to launch my game or program? Why are the non-ASCII characters in my script displaying or sending incorrectly? Why don't Hotstrings, Send, and Click work in certain games? How can performance be improved for games or at other times when the CPU is under heavy load? My antivirus program flagged AutoHotkey or a compiled script as malware. Is it really a virus? Common Tasks Where can I find the official build, or older releases? Can I run AHK from a USB drive? How can the output of a command line operation be retrieved? How can a script close, pause, suspend or reload other script(s)? How can a repeating action be stopped without exiting the script? How can context sensitive help for AutoHotkey functions be used in any editor? How to detect when a web page is finished loading? How can dates and times be compared or manipulated? How can I send the current Date and/or Time? How can I send text to a window which isn't active or isn't visible? How can Winamp be controlled even when it isn't active? How can MsgBox's button names be changed? How can I change the default editor, which is accessible via context menu or tray icon? How can I save the contents of my GUI controls? Can I draw something with AHK? How can I start an action when a window appears, closes or becomes [in]active? Hotkeys, Hotstrings, and Remapping How do I put my hotkeys and hotstrings into effect automatically every time I start my PC? I'm having trouble getting my mouse buttons working as hotkeys. Any advice? How can Tab and Space be defined as hotkeys? How can keys or mouse buttons be remapped so that they become different keys? How do I detect the double press of a key or button? How can a hotkey or hotstring be made exclusive to certain program(s)? In other words, I want a certain key to act as it normally does except when a specific window is active. How can a prefix key be made to perform its native function rather than doing nothing? How can the built-in Windows shortcut keys, such as Win+U (Utility Manager) and Win+R (Run), be changed or disabled? Can I use wildcards or regular expressions in Hotstrings? How can I use a hotkey that is not in my keyboard layout? My keypad has a special 000 key. Is it possible to turn it into a hotkey? General Troubleshooting What can I do if AutoHotkey won't install? Note: This answer applies only to the AutoHotkey v1 installer. The v2 installer works differently and has not been finalized. 7-zip Error: Use 7-zip or a compatible program to extract the setup files from the installer EXE, then run setup.exe or Installer.ahk (drag and drop Installer.ahk onto AutoHotkeyU32.exe). AutoHotkey's installer comes packaged as a 7-zip self-extracting archive which attempts to extract to the user's Temp directory and execute a compiled script. Sometimes system policies or other factors prevent the files from being extracted or executed. Usually in such cases the message "7-zip Error" is displayed. Manually extracting the files to a different directory may help. Setup hangs: If the setup window comes up blank or not at all, try one or both of the following: Hold Ctrl or Shift when the installer starts. If you get a UAC prompt, hold Ctrl or Shift as you click Yes/Continue. You should get a prompt asking whether you want to install with default options. Install using command line options. If you have manually extracted the setup files from the installer EXE, use either setup.exe /S or AutoHotkeyU32.exe Installer.ahk /S. Other: The suggestions above cover the most common problems. For further assistance, post on the forums. How do I restore the right-click context menu options for .ahk files? Normally if AutoHotkey is installed, right-clicking an AutoHotkey script (.ahk) file should give the following options: Run script Compile script (if Ahk2Exe is installed) Edit script Run as administrator Run with UI access (if the prerequisites are met) Note: On Windows 11, some of these options are usually relegated to a submenu that can be accessed by selecting "Show more options". Sometimes these options are overridden by settings in the current user's profile, such as if Open With has been used to change the default program for opening .ahk files. This can be fixed by deleting the following registry key: HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.ahk\UserChoice This can be done by applying this registry patch or by running UX\reset-assoc.ahk from the AutoHotkey installation directory. It may also be necessary to repair the default registry values, either by reinstalling AutoHotkey or by running UX\install.ahk from the AutoHotkey installation directory. Why doesn't my script work on Windows xxx even though it worked on a previous version? There are many variations of this problem, such as: I've upgraded my computer/Windows and now my script won't work. Hotkeys/hotstrings don't work when a program running as admin is active. Some windows refuse to be automated (e.g. Device Manager ignores Send). If you've switched operating systems, it is likely that something else has also changed and may be affecting your script. For instance, if you've got a new computer, it might have different drivers or other software installed. If you've also updated to a newer version of AutoHotkey, find out which version you had before and then check the changelog and compatibility notes. Also refer to the following question: How do I work around problems caused by User Account Control (UAC)? By default, User Account Control (UAC) protects "elevated" programs (that is, programs which are running as admin) from being automated by non-elevated programs, since that would allow them to bypass security restrictions. Hotkeys are also blocked, so for instance, a non-elevated program cannot spy on input intended for an elevated program. UAC may also prevent SendPlay and BlockInput from working. Common workarounds are as follows: Recommended: Run with UI access. This requires AutoHotkey to be installed under Program Files. There are several ways to do this, including: Right-click the script in Explorer and select Run with UI access. Use the UIAccess shell verb with Run, as in Run '*UIAccess "Your script.ahk"'. Use a command line such as "AutoHotkey32_UIA.exe" "Your script.ahk" (but include full paths where necessary). Set scripts to run with UI access by default, by checking the appropriate box in the Launch Settings GUI. Run the script as administrator. Note that this also causes any programs launched by the script to run as administrator, and may require the user to accept an approval prompt when launching the script. Disable the local security policy "Run all administrators in Admin Approval Mode" (not recommended). Disable UAC completely. This is not recommended, and is not feasible on Windows 8 or later. I can't edit my script via tray icon because it won't start due to an error. What should I do? You need to fix the error in your script before you can get your tray icon back. But first, you need to find the script file. Look for AutoHotkey.ahk in the following directories: Your Documents (or My Documents) folder. The directory where you installed AutoHotkey, usually C:\Program Files\AutoHotkey. If you are using AutoHotkey without having installed it, look in the directory which contains AutoHotkey.exe. If you are running another AutoHotkey executable directly, the name of the script depends on the executable. For example, if you are running AutoHotkey32.exe, look for AutoHotkey32.ahk. Note that depending on your system settings the ".ahk" part may be hidden, but the file should have an icon like You can usually edit a script file by right clicking it and selecting Edit Script. If that doesn't work, you can open the file in Notepad or another editor. If you launch AutoHotkey from the Start menu or by running AutoHotkey.exe directly (without command line parameters), it will look for a script in one of the locations shown above. Alternatively, you can create a script file (something.ahk) anywhere you like, and run the script file instead of running AutoHotkey. See also Command Line Parameter "Script Filename" and Portability of AutoHotkey.exe. How can I find and fix errors in my code? For simple scripts, see Debugging a Script. To show contents of a variable, use MsgBox or ToolTip. For complex scripts, see Interactive Debugging. Why is the Run function unable to launch my game or program? Some programs need to be started in their own directories (when in doubt, it is usually best to do so). For example: Run A_ProgramFiles "\Some Application\App.exe", A_ProgramFiles "\Some Application" If the program you are trying to start is in A_WinDir "\System32" and you are using AutoHotkey 32-bit on a 64-bit system, the File System Redirector may be interfering. To work around this, use A_WinDir

"\SysNative" instead; this is a virtual directory only visible to 32-bit programs running on 64-bit systems. Why are the non-ASCII characters in my script displaying or sending incorrectly? Short answer: Save the script as UTF-8. Non-ASCII characters are represented by different binary values depending on the chosen encoding. In order for such characters to be interpreted correctly, your text editor and AutoHotkey must be on the same codepage, so to speak. AutoHotkey v2 defaults to UTF-8 for all script files, although this can be overridden with the /CP command line switch. To save as UTF-8 in Notepad, select UTF-8 or UTF-8 with BOM from the Encoding drop-down in the Save As dialog. Note that Notepad in Windows 10 version 1903 and later defaults to UTF-8. To read other UTF-8 files which lack a byte order mark, use FileEncoding "UTF-8-RAW", the *P65001 option with FileRead, or "UTF-8-RAW" for the third parameter of FileOpen. The -RAW suffix can be omitted, but in that case any newly created files will have a byte order mark. Note that INI files accessed with the standard INI functions do not support UTF-8; they must be saved as ANSI or UTF-16. Why don't Hotstrings, Send, and Click work in certain games? Not all games allow AHK to send keys and clicks or receive pixel colors. But there are some alternatives, try all the solutions mentioned below. If all these fail, it may not be possible for AHK to work with your game. Sometimes games have a hack and cheat prevention measure, such as GameGuard and Hackshield. If they do, there is a high chance that AutoHotkey will not work with that game. Use SendPlay via the SendPlay function, SendMode Play and/or the hotstring option SP. SendPlay "abc" SendMode "Play" Send "abc" :SP:btw::by the way ; or #Hotstring SP ::btw::by the way Note: SendPlay may have no effect at all if User Account Control is enabled, even if the script is running as an administrator. Increase SetKeyDelay. For example: SetKeyDelay 0, 50 SetKeyDelay 0, 50, "Play" Try ControlSend, which might work in cases where the other Send modes fail: ControlSend "abc",, GameTitle Try the down and up event of a key with the various send methods: Send "{KEY Down} {KEY Up}" Try the down and up event of a key with a Sleep between them: Send "{KEY down}" Sleep 10 ; Try various milliseconds. Send "{KEY up}" How can performance be improved for games or at other times when the CPU is under heavy load? If a script's Hotkeys, Clicks, or Sends are noticeably slower than normal while the CPU is under heavy load, raising the script's process-priority may help. To do this, include the following line near the top of the script: ProcessSetPriority "High" My antivirus program flagged AutoHotkey or a compiled script as malware. Is it really a virus? Although it is certainly possible that the file has been infected, most often these alerts are false positives, meaning that the antivirus program is mistaken. One common suggestion is to upload the file to an online service such as virustotal or Jotti and see what other antivirus programs have to say. If in doubt, you could send the file to the vendor of your antivirus software for confirmation. This might also help us and other AutoHotkey users, as the vendor may confirm it is a false positive and fix their product to play nice with AutoHotkey. False positives might be more common for compiled scripts which have been compressed, such as with UPX (default for AutoHotkey 1.0 but not 1.1) or MPRESS (optional for AutoHotkey 1.1). As the default AutoHotkey installation does not include a compressor, compiled scripts are not compressed by default. Common Tasks Where can I find the official build, or older releases? See download page of AutoHotkey. Can I run AHK from a USB drive? See Portability of AutoHotkey.exe. How can the output of a command line operation be retrieved? Testing shows that due to file caching, a temporary file can be very fast for relatively small outputs. In fact, if the file is deleted immediately after use, it often does not actually get written to disk. For example: RunWait A_ComSpec ' /c dir > C:\My Temp File.txt' VarToContainContents := FileRead("C:\My Temp File.txt") FileDelete "C:\My Temp File.txt" To avoid using a temporary file (especially if the output is large), consider using the Shell.Exec() method as shown in the examples for the Run function. How can a script close, pause, suspend or reload other script(s)? First, here is an example that closes another script: DetectHiddenWindows True ; Allows a script's hidden main window to be detected. SetTitleMatchMode 2 ; Avoids the need to specify the full path of the file below. WinClose "ScriptFileName.ahk - AutoHotkey" ; Update this to reflect the script's name (case sensitive). To suspend, pause or reload another script, replace the last line above with one of these: PostMessage 0x0111, 65305,,, "ScriptFileName.ahk - AutoHotkey" ; Suspend. PostMessage 0x0111, 65306,,, "ScriptFileName.ahk - AutoHotkey" ; Pause. PostMessage 0x0111, 65303,,, "ScriptFileName.ahk - AutoHotkey" ; Reload. How can a repeating action be stopped without exiting the script? To pause or resume the entire script at the press of a key, assign a hotkey to the Pause function as in this example: ^!p::Pause ; Press Ctrl+Alt+P to pause. Press it again to resume. To stop an action that is repeating inside a Loop, consider the following working example, which is a hotkey that both starts and stops its own repeating action. In other words, pressing the hotkey once will start the Loop. Pressing the same hotkey again will stop it. #MaxThreadsPerHotkey 3 #z:: ; Win+Z hotkey (change this hotkey to suit your preferences). { static KeepWinZRunning := false if KeepWinZRunning ; This means an underlying thread is already running the loop below. { KeepWinZRunning := false ; Signal that thread's loop to stop. return ; End this thread so that the one underneath will resume and see the change made by the line above. } ; Otherwise: KeepWinZRunning := true Loop { ; The next four lines are the action you want to repeat (update them to suit your preferences): ToolTip "Press Win-Z again to stop this from flashing." Sleep 1000 ToolTip Sleep 1000 ; But leave the rest below unchanged. if not KeepWinZRunning ; The user signaled the loop to stop by pressing Win-Z again. break ; Break out of this loop. } KeepWinZRunning := false ; Reset in preparation for the next press of this hotkey. } #MaxThreadsPerHotkey 1 How can context sensitive help for AutoHotkey functions be used in any editor? Rajat created this script. How to detect when a web page is finished loading? With Internet Explorer, perhaps the most reliable method is to use DllCall and COM as demonstrated at www.autohotkey.com/forum/topic19256.html. On a related note, the contents of the address bar and status bar can be retrieved as demonstrated at www.autohotkey.com/forum/topic19255.html. Older, less reliable method: The technique in the following example will work with MS Internet Explorer for most pages. A similar technique might work in other browsers: Run "www.yahoo.com" MouseMove 0, 0 ; Prevents the status bar from showing a mouse-hover link instead of "Done". WinWait "Yahoo! - " WinActivate if StatusBarWait("Done", 30) MsgBox "The page is done loading." else MsgBox "The wait timed out or the window was closed." How can dates and times be compared or manipulated? The DateAdd function can add or subtract a quantity of days, hours, minutes, or seconds to a time-string that is in the YYYYMMDDHH24MISS format. The following example subtracts 7 days from the specified time: Result := DateAdd (VarContainingTimestamp, -7, "days") To determine the amount of time between two dates or times, see DateDiff, which gives an example. Also, the built-in variable A_Now contains the current local time. Finally, there are several built-in date/time variables, as well as the FormatTime function to create a custom date/time string. How can I send the current Date and/or Time? Use FormatTime or built-in variables for date and time. How can I send text to a window which isn't active or isn't visible? Use ControlSend. How can Winamp be controlled even when it isn't active? See Automating Winamp. How can MsgBox's button names be changed? Here is an example. How can I change the default editor, which is accessible via context menu or tray icon? In the example section of Edit you will find a script that allows you to change the default editor. How can I save the contents of my GUI controls? Use Gui.Submit. For Example: MyGui := Gui() MyGui.Add ("Text",, "Enter some Text and press Submit:") MyGui.Add("Edit", "vMyEdit") MyGui.Add("Button",, "Submit").OnEvent("Click", Submit) MyGui.Show Submit(*) { Saved := MyGui.Submit(false) MsgBox "Content of the edit control: " Saved.MyEdit } Can I draw something with AHK? See GDI+ standard library by tic. It's also possible with some rudimentary methods using Gui, but in a limited way. How can I start an action when a window appears, closes or becomes [in] active? Use WinWait, WinWaitClose or WinWait[Not]Active. There are also user-created solutions such as OnWin.ahk and [How to] Hook on to Shell to receive its messages. Hotkeys, Hotstrings, and Remapping How do I put my hotkeys and hotstrings into effect automatically every time I start my PC? There are several ways to make a script (or any program) launch automatically every time you start your PC. The easiest is to place a shortcut to the script in the Startup folder: Find the script file, select it, and press Ctrl+C. Press Win+R to open the Run dialog, then enter shell:startup and click OK or Enter. This will open the Startup folder for the current user. To instead open the folder for all users, enter shell:common startup (however, in that case you must be an administrator to proceed). Right click inside the window, and click "Paste Shortcut". The shortcut to the script should now be in the Startup folder. I'm having trouble getting my mouse buttons working as hotkeys. Any advice? The left and right mouse buttons should be assignable normally (for example, #LButton:: is the Win+LeftButton hotkey). Similarly, the middle button and the turning of the mouse wheel should be assignable normally except on mice whose drivers directly control those buttons. The fourth button (XButton1) and the fifth button (XButton2) might be assignable if your mouse driver allows their clicks to be seen by the system. If they cannot be seen -- or if your mouse has more than five buttons that you want to use -- you can try configuring the software that came with the mouse (sometimes accessible in the Control Panel or Start Menu) to send a keystroke whenever you press one of these buttons. Such a keystroke can then be defined as a hotkey in a script. For example, if you configure the fourth button to send Ctrl+F1, you can then indirectly configure that button as a hotkey by using ^F1:: in a script. If you have a five-button mouse whose fourth and fifth buttons cannot be seen, you can try changing your mouse driver to the default driver included with the OS. This assumes there is such a driver for your particular mouse and that you can live without the features provided by your mouse's custom software. How can Tab and Space be defined as hotkeys? Use the names of the keys (Tab and Space) rather than their characters. For example, #Space is Win+Space and ^!Tab is Ctrl+Alt+Tab. How can keys or mouse buttons be remapped so that they become different keys? This is described on the remapping page. How do I detect the double press of a key or button? Use built-in variables for hotkeys as follows: ~Ctrl:: { if (ThisHotkey = A_PriorHotkey && A_TimeSincePriorHotkey < 200) MsgBox "double-press" } How can a hotkey or hotstring be made exclusive to certain program(s)? In other words, I want a certain key to act as it normally does except when a specific window is active. The preferred method is #HotIf. For example: #HotIf WinActive("ahk_class Notepad") ^a::MsgBox "You pressed Control-A while Notepad is active." How can a prefix key be made to perform its native function rather than doing nothing? Consider the following example, which makes Numpad0 into a prefix key: Numpad0 & Numpad1::MsgBox "You pressed Numpad1 while holding down Numpad0." Now, to make Numpad0 send a real Numpad0 keystroke whenever it wasn't used to launch a hotkey such as the above, add the following hotkey: $Numpad0::Send "{Numpad0}" The $ prefix is needed to prevent a warning dialog about an infinite loop (since the hotkey "sends itself"). In addition, the above action occurs at the time the key is released. How can the built-in Windows shortcut keys, such as Win+U (Utility Manager) and Win+R (Run), be changed or disabled? Here are some examples. Can I use wildcards or regular expressions in Hotstrings? Use the script by polyethene (examples are included). How can I use a hotkey that is not in my keyboard layout? See Special Keys. My keypad has a special 000 key. Is it possible to turn it into a hotkey? You can. This example script makes 000 into an equals key. You can change the action by replacing the Send "=" line with line(s) of your choice. Functions - Definition & Usage | AutoHotkey v2 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. Advanced Hotkey Features | AutoHotkey v2 Advanced Hotkey Features Table of Contents General Remap easy to reach but rarely used keys Use any keys as modifiers Make the mouse wheel perform alt-tabbing Make a keyboard key become a mouse button Make your hotkeys context-sensitive Define abbreviations that expand as you type them Gaming Reduce wear and tear on your fingers Create mouse hotkeys Create "pass-through" hotkeys Automate game actions on the screen Use the keyboard hook Related Topics General Remap easy to reach but rarely used keys Some of the easiest keys to reach on the keyboard are also the least frequently used. Make these keys do something useful! For example, if you rarely use the right Alt, make it perform the action you do most often: RAlt::MsgBox "You pressed the right ALT key." You

can even do the above without losing the native function of the right Alt by assigning the right Alt to be a "prefix" for at least one other hotkey. In the below example, the right Alt has become a prefix, which automatically allows it to modify all other keys as it normally would. But if you press and release the right Alt without having used it to modify another key, its hotkey action (above) will take effect immediately: RAlt & j::AltTab Use any keys as modifiers Don't be limited to using only Ctrl, Alt, Shift, and Win as modifiers; you can combine any two keys or mouse buttons to form a custom hotkey. For example: Hold down Numpad0 and press Numpad1 to launch a hotkey (Numpad0 & Numpad1::); hold down CapsLock and press another key, or click a mouse button (CapsLock & RButton::). In this case, CapsLock's state (on or off) is not changed when it is used to launch the hotkey. For details, see custom combinations of keys. Make the mouse wheel perform alt-tabbing Convert the mouse wheel (or any other keys of your choice) into a complete substitute for Alt-Tab. Click the wheel to show or hide the menu, and turn it to navigate through the menu. The wheel will still function normally whenever the Alt-Tab menu isn't visible. Syntax: MButton::AltTabMenu WheelDown::AltTab WheelUp::ShiftAltTab Make a keyboard key become a mouse button Make a keyboard key become a mouse button, or have an action repeated continuously while you're holding down a key or mouse button. See the remapping page for examples. Make your hotkeys context-sensitive Have your easiest-to-reach hotkeys perform an action appropriate to the type of window you're working with. In the following example, the right Ctrl performs a different action depending on whether Notepad or Calculator is the active window: #HotIf WinActive("ahk_class Notepad") RControl::Send "^s" ; Save the current file in Notepad. #HotIf WinActive("Calculator") RControl::Send "^c!{tab}^v" ; Copy the Calculator's result into the previously active window. See #HotIf for details. Define abbreviations that expand as you type them Also known as hotstrings. No special training or scripting experience is needed. For example, a script containing the following lines would expand ceo, cfo, and btw wherever you type them: ::ceo::Chief Executive Officer ::cfo::Chief Financial Officer ::btw::by the way Gaming Reduce wear and tear on your fingers Reduce wear and tear on your fingers by using virtually any key as a hotkey, including single letters, arrow keys, Numpad keys, and even the modifier keys themselves (Ctrl, Alt, Win, and Shift). Create mouse hotkeys Create mouse hotkeys, including the mouse wheel button (MButton) and the turning of the wheel up/down or left/right (WheelUp, WheelDown, WheelLeft, and WheelRight). You can also combine a keyboard key with a mouse button. For example, control-right-button would be expressed as ^RButton::. Create "pass-through" hotkeys For example, the left mouse button can trigger a hotkey action even while the click itself is being sent into the game normally (syntax: ~LButton::). Automate game actions on the screen Use functions such as PixelSearch, PixelGetColor, and ImageSearch to automate game actions. Use the keyboard hook Have the option of using the keyboard hook to implement hotkeys, which might be more responsive than other hotkey methods while the CPU is under load in a game. The hook might also be able to override any restrictions a game may have about which keys can be "mapped" to game actions. Related Topics Hotkeys Hotstrings Remapping Keys Hotkeys - Definition & Usage | AutoHotkey v2 Hotkeys (Mouse, Joystick and Keyboard Shortcuts) Table of Contents Introduction and Simple Examples Hotkey Modifier Symbols Context-sensitive Hotkeys Custom Combinations Other Features Mouse Wheel Hotkeys Hotkey Tips and Remarks Alt-Tab Hotkeys Named Function Hotkeys 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. 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." <^::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. Hotstrings - Definition & Usage | AutoHotkey v2 Hotstrings Table of Contents Introduction and Simple Examples Ending Characters Options Long Replacements Context-sensitive Hotstrings AutoCorrect Remarks Named Function Hotstrings Hotstring Helper 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. Quick Reference | AutoHotkey v2 Version 2.0.2 https://www.autohotkey.com © 2014 Steve Gray, Chris Mallett, portions © AutoIt Team and various others Software License: GNU General Public License Quick Reference Getting started: How to use the program Tutorials: How to run example code How to write hotkeys How to send keystrokes How to run programs How to manage windows Beginner tutorial by tidbit Text editors with AutoHotkey support Frequently asked questions Scripts: Concepts and conventions: explanations of various things you need to know. Scripting language: how to write scripts. Miscellaneous topics List of built-in functions Variables and expressions How to use functions Objects Interactive debugging Keyboard and mouse: Hotkeys (mouse, joystick and keyboard shortcuts) Hotstrings and auto-replace Remapping keys and buttons List of keys, mouse buttons and joystick controls Other: DllCall RegEx quick reference Acknowledgements A special thanks to Jonathan Bennett, whose generosity in releasing AutoIt v2 as free software in 1999 served as an inspiration and time-saver for myself and many others worldwide. In addition, many of AutoHotkey's enhancements to the AutoIt v2 command set, as well as the Window Spy and the old script compiler, were adapted directly from the AutoIt v3 source code. So thanks to Jon and the other AutoIt authors for those as well. Finally, AutoHotkey would not be what it is today without these other individuals. ~ Chris Mallett List of Keys (Keyboard, Mouse and Joystick) | AutoHotkey v2 List of Keys (Keyboard, Mouse and Joystick) Table of Contents Mouse General Buttons Advanced Buttons Wheel Keyboard General Keys Cursor Control Keys Numpad Keys Function Keys Modifier Keys Multimedia Keys Other Keys Joystick Hand-held Remote Controls Special Keys 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). WheelLeftWheelRight 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 / NumpadIns0 / Ins Numpad1 / NumpadEnd1 / End Numpad2 / NumpadDown2 / ? Numpad3 / NumpadPgDn3 / PgDn Numpad4 / NumpadLeft4 / ? Numpad5 / NumpadClear5 / typically does nothing Numpad6 / NumpadRight6 / ? Numpad7 / NumpadHome7 / Home Numpad8 / NumpadUp8 / ? Numpad9 / NumpadPgUp9 / 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. 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 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}" } Scripting Language | AutoHotkey v2 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 General Conventions Comments Expressions Strings / Text Variables Constants Operators Function Calls Function Call Statements Optional Parameters Operators for Objects 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: NameValueTypeDescription False0IntegerBoolean false, sometimes meaning "off", "no", etc. True1IntegerBoolean 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", , , ¬epadPID) 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. License | AutoHotkey v2 Software License GNU GENERAL PUBLIC LICENSE Version 2, June 1991 Copyright (C) 1989, 1991 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too. When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things. To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights. We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software. Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations. Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all. The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does. 1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program. You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee. 2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions: a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change. b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be

licensed as a whole at no charge to all third parties under the terms of this License. c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.) These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program. In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License. 3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following: a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or, b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or, c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.) The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable. If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code. 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance. 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it. 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License. 7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program. If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances. It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice. This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License. 8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License. 9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation. 10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally. NO WARRANTY 11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. END OF TERMS AND CONDITIONS Objects - Definition & Usage | AutoHotkey v2 Objects An object combines a number of properties and methods. Related topics: Objects: General explanation of objects. Object Protocol: Specifics about how a script interacts with an object. Function objects: Objects which can be called. IsObject can be used to determine if a value is an object: Result := IsObject(expression) See Built-in Classes for a list of standard object types. There are two fundamental types: AutoHotkey objects are instances of the Object class. These support ad hoc properties, and have methods for discovering which properties exist. Array, Map and all user defined and built-in classes are derived from Object. COM objects such as those created by ComObject. These are implemented by external libraries, so often differ in behaviour to AutoHotkey objects. ComObject typically represents a COM or "Automation" object implementing the IDispatch interface, but is also used to wrap values of specific types to be passed to COM objects and functions. Table of Contents Basic Usage - Arrays, Maps (Associative Arrays), Objects, Freeing Objects Extended Usage - Arrays of Arrays Custom Objects - Creating a Base Object, Classes, Construction and Destruction, Meta-Functions Primitive Values Implementation - Reference-Counting, Pointers to Objects Basic Usage Arrays Create an Array: MyArray := [Item1, Item2, ..., ItemN] MyArray := Array(Item1, Item2, ..., ItemN) Retrieve an item (or array element): Value := MyArray[Index] Change the value of an item (Index must be between 1 and Length, or an equivalent reverse index): MyArray[Index] := Value Insert one or more items at a given index using the InsertAt method: MyArray.InsertAt(Index, Value, Value2, ...) Append one or more items using the Push method: MyArray.Push(Value, Value2, ...) Remove an item using the RemoveAt method: RemovedValue := MyArray.RemoveAt(Index) Remove the last item using the Pop method: RemovedValue := MyArray.Pop() Length returns the number of items in the array. Looping through an array's contents can be done either by index or with a For-loop. For example: MyArray := ["one", "two", "three"] ; Iterate from 1 to the end of the array: Loop MyArray.Length MsgBox MyArray[A_Index] ; Enumerate the array's contents: For index, value in MyArray MsgBox "Item " index " is '" value "'" ; Same thing again: For value in MyArray MsgBox "Item " A_Index " is '" value "'" Maps (Associative Arrays) A Map or associative array is an object which contains a collection of unique keys and a collection of values, where each key is associated with one value. Keys can be strings, integers or objects, while values can be of any type. An associative array can be created as follows: MyMap := Map("KeyA", ValueA, "KeyB", ValueB, ..., "KeyZ", ValueZ) Retrieve an item, where Key is a variable or expression: Value := MyMap[Key] Assign an item: MyMap[Key] := Value Remove an item using the Delete method: RemovedValue := MyMap.Delete(Key) Enumerating items: MyMap := Map("ten", 10, "twenty", 20, "thirty", 30) For key, value in MyMap MsgBox key ' = ' value Objects An object can have properties and items (such as array elements). Items are accessed using [] as shown in the previous sections. Properties are usually accessed by writing a dot

followed by an identifier (just a name). Methods are properties which can be called. Examples: Retrieve or set a property literally named Property: Value := MyObject.Property MyObject.Property := Value Retrieve or set a property where the name is determined by evaluating an expression or variable: Value := MyObject.%Expression% MyObject.%Expression% := Value Call a property/method literally named Method: ReturnValue := MyObject.Method(Parameters) Call a property/method where the name is determined by evaluating an expression or variable: ReturnValue := MyObject.%Expression%(Parameters) Sometimes parameters are accepted when retrieving or assigning properties: Value := MyObject.Property[Parameters] MyObject.Property[Parameters] := Value An object may also support indexing: MyArray[Index] actually invokes the __Item property of MyArray, passing Index as a parameter. Object Literal An object literal can be used within an expression to create an improvised object. An object literal consists of a pair of braces ({}) enclosing a list of comma-delimited name-value pairs. Each pair consists of a literal (unquoted) property name and a value (sub-expression) separated by a colon (:). For example: Coord := {X: 13, Y: 240} This is equivalent: Coord := Object() Coord.X := 13 Coord.Y := 240 Each name-value pair causes a value property to be defined, with the exception that Base can be set (with the same restrictions as a normal assignment). Name substitution allows a property name to be determined by evaluating an expression or variable. For example: parts := StrSplit("key = value", "=", " ") pair := {%parts[1]%: parts[2]} MsgBox pair.key Freeing Objects Scripts do not free objects explicitly. When the last reference to an object is released, the object is freed automatically. A reference stored in a variable is released automatically when that variable is assigned some other value. For example: obj := {} ; Creates an object. obj := "" ; Releases the last reference, and therefore frees the object. Similarly, a reference stored in a property or array element is released when that property or array element is assigned some other value or removed from the object. arr := [{}] ; Creates an array containing an object. arr[1] := {} ; Creates a second object, implicitly freeing the first object. arr.RemoveAt(1) ; Removes and frees the second object. Because all references to an object must be released before the object can be freed, objects containing circular references aren't freed automatically. For instance, if x.child refers to y and y.parent refers to x, clearing x and y is not sufficient since the parent object still contains a reference to the child and vice versa. To resolve this situation, remove the circular reference. x := {}, y := {} ; Create two objects. x.child := y, y.parent := x ; Create a circular reference. y.parent := "" ; The circular reference must be removed before the objects can be freed. x := "", y := "" ; Without the above line, this would not free the objects. For more advanced usage and details, see Reference Counting. Extended Usage Arrays of Arrays Although "multi-dimensional" arrays are not supported, a script can combine multiple arrays or maps. For example: grid := [[1,2,3], [4,5,6], [7,8,9]] MsgBox grid[1][3] ; 3 MsgBox grid[3][2] ; 8 A custom object can implement multi-dimensional support by defining an __Item property. For example: class Array2D extends Array { __new(x, y) { this.Length := x * y this.Width := x this.Height := y } __Item[x, y] { get => super[this.Width * (y-1) + x] set => super[this.Width * (y-1) + x] := value } } grid := Array2D(4, 3) grid[4, 1] := "#" grid[3, 2] := "#" grid[2, 2] := "#" grid[1, 3] := "#" gridtext := "" Loop grid.Height { y := A_Index Loop grid.Width { x := A_Index gridtext .= grid[x, y] || "-" } gridtext .= "`n" } MsgBox gridtext A real script should perform error-checking and override other methods, such as __Enum to support enumeration. Custom Objects There are two general ways to create custom objects: Ad hoc: create an object and add properties. Delegation: define properties in a shared base object or class. Meta-functions can be used to further control how an object behaves. Note: Within this section, an object is any instance of the Object class. This section does not apply to COM objects. Ad Hoc Properties and methods (callable properties) can generally be added to new objects at any time. For example, an object with one property and one method might be constructed like this: ; Create an object. thing := {} ; Store a value. thing.foo := "bar" ; Define a method. thing.test := thing_test ; Call the method. thing.test() thing_test(this) { MsgBox this.foo } You could similarly create the above object with thing := {foo: "bar"}. When using the {property:value} notation, quote marks must not be used for properties. When thing.test() is called, thing is automatically inserted at the beginning of the parameter list. By convention, the function is named by combining the "type" of object and the method name, but this is not a requirement. In the example above, test could be assigned some other function or value after it is defined, in which case the original function is lost and cannot be called via this property. An alternative is to define a read-only method, as shown below: thing.DefineProp 'test', {call: thing_test} See also: DefineProp Delegation Objects are prototype-based. That is, any properties not defined in the object itself can instead be defined in the object's base. This is known as inheritance by delegation or differential inheritance, because an object can implement only the parts that make it different, while delegating the rest to its base. Although a base object is also generally known as a prototype, we use "a class's Prototype" to mean the object upon which every instance of the class is based, and "base" to mean the object upon which an instance is based. AutoHotkey's object design was influenced primarily by JavaScript and Lua, with a little C#. We use obj.base in place of JavaScript's obj.__proto__ and cls.Prototype in place of JavaScript's func.prototype. (Class objects are used in place of constructor functions.) An object's base is also used to identify its type or class. For example, x := [] creates an object based on Array.Prototype, which means that the expressions x is Array and x.HasBase(Array.Prototype) are true, and type(x) returns "Array". Each class's Prototype is based on the Prototype of its base class, so x.HasBase(Object.Prototype) is also true. Any instance of Object or a derived class can be a base object, but an object can only be assigned as the base of an object with the same native type. This is to ensure that built-in methods can always identify the native type of an object, and operate only on objects that have the correct binary structure. Base objects can be defined two different ways: By creating a normal object. By defining a class. Each class has a Prototype property containing an object which all instances of that class are based on, while the class itself becomes the base object of any direct subclasses. A base object can be assigned to the base property of another object, but typically an object's base is set implicitly when it is created. Creating a Base Object Any object can be used as the base of any other object which has the same native type. The following example builds on the previous example under Ad Hoc (combine the two before running it): other := {} other.base := thing other.test() In this case, other inherits foo and test from thing. This inheritance is dynamic, so if thing.foo is modified, the change will be reflected by other.foo. If the script assigns to other.foo, the value is stored in other and any further changes to thing.foo will have no effect on other.foo. When other.test() is called, its this parameter contains a reference to other instead of thing. Classes In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). Wikipedia In more general terms, a class is a set or category of things having some property or attribute in common. In AutoHotkey, a class defines properties to be shared by instances of the class (and methods, which are callable properties). An instance is just an object which inherits properties from the class, and can typically also be identified as belonging to that class (such as with the expression instance is ClassName). Instances are typically created by calling ClassName(). Since Objects are dynamic and prototype-based, each class consists of two parts: The class has a Prototype object, on which all instances of the class are based. All methods and dynamic properties that apply to instances of the class are contained by the prototype object. This includes all properties and methods which lack the static keyword. The class itself is an object, containing only static methods and properties. This includes all properties and methods with the static keyword, and all nested classes. These do not apply to a specific instance, and can be used by referring to the class itself by name. The following shows most of the elements of a class definition: class ClassName extends BaseClassName { InstanceVar := Expression static ClassVar := Expression class NestedClass { ... } Method() { ... } static Method() { ... } Property[Parameters] ; Use brackets only when parameters are present. { get { return value of property } set { Store or otherwise handle value } } ShortProperty { get => Expression which calculates property value set => Expression which stores or otherwise handles value } ShorterProperty => Expression which calculates property value } When the script is loaded, this constructs a Class object and stores it in a global constant (read-only variable) with the name ClassName. If extends BaseClassName is present, BaseClassName must be the full name of another class. The full name of each class is stored in ClassName.Prototype.__Class. Because the class itself is accessed through a variable, the class name cannot be used to both reference the class and create a separate variable (such as to hold an instance of the class) in the same context. For example, box := Box() will not work, because box and Box both resolve to the same thing. Attempting to reassign a top-level (not nested) class in this manner results in a load time error. Within this documentation, the word "class" on its own usually means a class object constructed with the class keyword. Class definitions can contain variable declarations, method definitions and nested class definitions. Instance Variables An instance variable is one that each instance of the class has its own copy of. They are declared and behave like normal assignments, but the this. prefix is omitted (only directly within the class body): InstanceVar := Expression These declarations are evaluated each time a new instance of the class is created with ClassName(), after all base-class declarations are evaluated but before __New is called. This is achieved by automatically creating a method named __Init containing a call to super.__Init() and inserting each declaration into it. Therefore, a single class definition must not contain both an __Init method and an instance variable declaration. Expression can access other instance variables and methods via this. Global variables may be read, but not assigned. An additional assignment (or use of the reference operator) within the expression will generally create a variable local to the __Init method. For example, x := y := 1 would set this.x and a local variable y (which would be freed once all initializers have been evaluated). To access an instance variable (even within a method), always specify the target object; for example, this.InstanceVar. Declarations like x.y := z are also supported, provided that x was previously defined in this class. For example, x := {}, x.y := 42 declares x and also initializes this.x.y. Static/Class Variables Static/class variables belong to the class itself, but their values can be inherited by subclasses. They are declared like instance variables, but using the static keyword: static ClassVar := Expression These declarations are evaluated only once, when the class is initialized. A static method named __Init is automatically defined for this purpose. Each declaration acts as a normal property assignment, with the class object as the target. Expression has the same interpretation as for instance variables, except that this refers to the class itself. To assign to a class variable anywhere else, always specify the class object; for example, ClassName.ClassVar := Value. If a subclass does not own a property by that name, Subclass.ClassVar can also be used to retrieve the value; so if the value is a reference to an object, subclasses will share that object by default. However, Subclass.ClassVar := y would store the value in Subclass, not in ClassName. Declarations like static x.y := z are also supported, provided that x was previously defined in this class. For example, static x := {}, x.y := 42 declares x and also initializes ClassName.x.y. Because Prototype is implicitly defined in each class, static Prototype.sharedValue := 1 can be used to set values which are dynamically inherited by all instances of the class (until shadowed by a property on the instance itself). Nested Classes Nested class definitions allow a class object to be associated with a static/class variable of the outer class instead of a separate global variable. In the example above, class NestedClass constructs a Class object and stores it in ClassName.NestedClass. Subclasses could inherit NestedClass or override it with their own nested class (in which case

(WhichClass.NestedClass)() could be used to instantiate whichever class is appropriate). class NestedClass { ... } Nesting a class does not imply any particular relationship to the outer class. The nested class is not instantiated automatically, nor do instances of the nested class have any connection with an instance of the outer class, unless the script explicitly makes that connection. However, due to the way methods work for Objects, WhichClass.NestedClass() implicitly passes WhichClass as the first parameter, equivalent to WhichClass.NestedClass.Call(WhichClass). Unless static Call() is overridden, this parameter is automatically passed to __New. Methods Method definitions look identical to function definitions. Each method definition creates a Func with a hidden first parameter named this, and defines a property which is used to call the method or retrieve its function object. There are two types of methods: Instance methods are defined as below, and are attached to the class's Prototype, which makes them accessible via any instance of the class. When the method is called, this refers to an instance of the class. Static methods are defined by preceding the method name with the separate keyword static. These are attached to the class object itself, but are also inherited by subclasses, so this refers to either the class itself or a subclass. The method definition below creates a property of the same type as target.DefineProp('Method', {call: funcObj}). By default, target.Method returns funcObj and attempting to assign to target.Method throws an error. These defaults can be overridden by defining a property or calling DefineProp. Method() { ... } Fat arrow syntax can be used to define a single-line method which returns an expression: Method() => Expression Super Inside a method or a property getter/setter, the keyword super can be used in place of this to access the superclass versions of methods or properties which are overridden in a derived class. For example, super.Method() in the class defined above would typically call the version of Method which was defined within BaseClassName. Note: super.Method() always invokes the base of the class or prototype object associated with the current method's original definition, even if this is derived from a subclass of that class or some other class entirely. super.Method() implicitly passes this as the first (hidden) parameter. Since it is not known where (or whether) ClassName exists within the chain of base objects, ClassName itself is used as the starting point. Therefore, super.Method() is mostly equivalent to (ClassName.Prototype.base.Method)(this) (but without Prototype when Method is static). However, ClassName.Prototype is resolved at load time. An error is thrown if the property is not defined in a superclass or cannot be invoked. The keyword super must be followed by one of the following symbols: .[(super() is equivalent to super.call(). Properties A property definition creates a dynamic property, which calls a method instead of simply storing or returning a value. Property[Parameters] { get { return property value } set { Store or otherwise handle value } } Property is simply the name of the property, which will be used to invoke it. For example, obj.Property would call get while obj.Property := value would call set. Within get or set, this refers to the object being invoked. Within set, value contains the value being assigned. Parameters can be defined by enclosing them in square brackets to the right of the property name, and are passed the same way - but they should be omitted when parameters are not present (see below). Aside from using square brackets, parameters of properties are defined the same way as parameters of methods - optional, ByRef and variadic parameters are supported. If a property is invoked with parameters but has none defined, parameters are automatically forwarded to the __Item property of the object returned by get. For example, this.Property[x] would have the same effect as (this.Property)[x] or y := this.Property, y[x]. Empty brackets (this.Property[]) always cause the __Item property of Property's value to be invoked, but a variadic call such as this.Property[args*] has this effect only if the number of parameters is non-zero. Static properties can be defined by preceding the property name with the separate keyword static. In that case, this refers to the class itself or a subclass. The return value of set is ignored. For example, val := obj.Property := 42 always assigns val := 42 regardless of what the property does, unless it throws an exception or exits the thread. Each class can define one or both halves of a property. If a class overrides a property, it can use super.Property to access the property defined by its base class. If Get or Set is not defined, it can be inherited from a base object. If Get is undefined, the property can return a value inherited from a base. If Set is undefined in this and all base objects (or is obscured by an inherited value property), attempting to set the property causes an exception to be thrown. A property definition with both get and set actually creates two separate functions, which do not share local or static variables or nested functions. As with methods, each function has a hidden parameter named this, and set has a second hidden parameter named value. Any explicitly defined parameters come after those. While a property definition defines the get and set accessor functions for a property in the same way as DefineProp, a method definition defines the call accessor function. Any class may contain a property definition and a method definition with the same name. If a property without a call accessor function (a method) is called, get is invoked with no parameters and the result is then called as a method. Fat Arrow Properties Fat arrow syntax can be used to define a property getter or setter which returns an expression: ShortProperty[Parameters] { get => Expression which calculates property value set => Expression which stores or otherwise handles value } When defining only a getter, the braces and get can be omitted: ShorterProperty[Parameters] => Expression which calculates property value In both cases, the square brackets must be omitted unless parameters are defined. __Enum Method __Enum(NumberOfVars) The __Enum method is called when the object is passed to a for-loop. This method should return an enumerator which will return items contained by the object, such as array elements. If left undefined, the object cannot be passed directly to a for-loop unless it has an enumerator-compatible Call method. NumberOfVars contains the number of variables passed to the for-loop. If NumberOfVars is 2, the enumerator is expected to assign the key or index of an item to the first parameter and the value to the second parameter. Each key or index should be accepted as a parameter of the __Item property. This enables DBGp-based debuggers to get or set a specific item after listing them by invoking the enumerator. __Item Property The __Item property is invoked when the indexing operator (array syntax) is used with the object. In the following example, the property is declared as static so that the indexing operator can be used on the Env class itself. For another example, see Array2D. class Env { static __Item[name] { get => EnvGet(name) set => EnvSet(name, value) } } Env["PATH"] .= ";" A_ScriptDir ; Only affects this script and child processes. MsgBox Env["PATH"] __Item is effectively a default property name (if such a property has been defined): object[params] is equivalent to object.__Item[params] when there are parameters. object[] is equivalent to object.__Item. For example: obj := {} obj[] := Map() ; Equivalent to obj.__Item := Map() obj["base"] := 10 MsgBox obj.base = Object.prototype ; True MsgBox obj["base"] ; 10 Note: When an explicit property name is combined with empty brackets, as in obj.prop[], it is handled as two separate operations: first retrieve obj.prop, then invoke the default property of the result. This is part of the language syntax, so is not dependent on the object. Construction and Destruction Whenever an object is created by the default implementation of ClassName(), the new object's __New method is called in order to allow custom initialization. Any parameters passed to ClassName() are forwarded to __New, so can affect the object's initial content or how it is constructed. When an object is destroyed, __Delete is called. For example: m1 := GMem(0, 10) m2 := {base: GMem.Prototype}, m2.__New(0, 30) ; Note: For general memory allocations, use Buffer() instead. class GMem { __New(aFlags, aSize) { this.ptr := DllCall("GlobalAlloc", "UInt", aFlags, "Ptr", aSize, "Ptr") if !this.ptr throw MemoryError() MsgBox "New GMem of " aSize " bytes at address " this.ptr "." } __Delete() { MsgBox "Delete GMem at address " this.ptr "." DllCall("GlobalFree", "Ptr", this.ptr) } } __Delete is not called for any object which owns a property named "__Class". Prototype objects have this property by default. If an exception or runtime error is thrown while __Delete is executing and is not handled within __Delete, it acts as though __Delete was called from a new thread. That is, an error dialog is displayed and __Delete returns, but the thread does not exit (unless it was already exiting). If the script is directly terminated by any means, including the tray menu or ExitApp, any functions which have yet to return do not get the chance to do so. Therefore, any objects referenced by local variables of those functions are not released, so __Delete is not called. When the script exits, objects contained by global and static variables are released automatically in an arbitrary, implementation-defined order. When __Delete is called during this process, some global or static variables may have already been released, but any references contained by the object itself are still valid. It is therefore best for __Delete to be entirely self-contained, and not rely on any global or static variables. Class Initialization Each class is initialized automatically when a reference to the class is evaluated for the first time. For example, if MyClass has not yet been initialized, MyClass.MyProp would cause the class to be initialized before the property is retrieved. Initialization consists of calling two static methods: __Init and __New. static __Init is defined automatically for every class, and always begins with a reference to the base class if one was specified, to ensure it is initialized. Static/class variables and nested classes are initialized in the order that they were defined, except when a nested class is referenced during initialization of a previous variable or class. If the class defines or inherits a static __New method, it is called immediately after __Init. It is important to note that __New may be called once for the class in which it is defined and once for each subclass which does not define its own (or which calls super.__New()). This can be used to perform common initialization tasks for each subclass, or modify subclasses in some way before they are used. If static __New is not intended to act on derived classes, that can be avoided by checking the value of this. In some cases it may be sufficient for the method to delete itself, such as with this.DeleteProp('__New'); however, the first execution of __New might be for a subclass if one is nested in the base class or referenced during initialization of a static/class variable. A class definition also has the effect of referencing the class. In other words, when execution reaches a class definition during script startup, __Init and __New are called automatically, unless the class was already referenced by the script. However, if execution is prevented from reaching the class definition, such as by return or an infinite loop, the class is initialized only if it is referenced. Once automatic initialization begins, it will not occur again for the same class. This is generally not a problem unless multiple classes refer to each other. For example, consider the two classes below. When A is initialized first, evaluating B.SharedArray (A1) causes B to be initialized before retrieving and returning the value, but A.SharedValue (A3) is undefined and does not cause initialization of A because it is already in progress. In other words, if A is accessed or initialized first, the order is A1 to A3; otherwise it is B1 to B4: MsgBox A.SharedArray.Length MsgBox B.SharedValue class A { static SharedArray := B.SharedArray ; A1 ; B3 static SharedValue := 42 ; B4 } class B { static SharedArray := StrSplit("XYZ") ; A2 ; B1 static SharedValue := A.SharedValue ; A3 (Error) ; B2 } Meta-Functions class ClassName { __Get(Name, Params) __Set(Name, Params, Value) __Call(Name, Params) } Name The name of the property or method. Params An Array of parameters. This includes only the parameters between () or [], so may be empty. The meta-function is expected to handle cases such as x.y[z] where x.y is undefined. Value The value being assigned. Meta-functions define what happens when an undefined property or method is invoked. For example, if obj.unk has not been assigned a value, it invokes the __Get meta-function. Similarly, obj.unk := value invokes __Set and obj.unk() invokes __Call. Properties and methods can be defined in the object itself or any of its base objects. In general, for a meta-function to be called for every property, one must avoid defining any properties. Built-in properties such as Base can be overridden with a property definition or DefineProp. If a meta-function is defined, it must perform whatever default action is required. For example, the following might be

expected: Call: Throw a MethodError. If parameters were given, throw an exception (there's no object to forward the parameters to). Get: Throw a PropertyError. Set: Define a new property with the given value, such as by calling DefineProp. Any callable object can be used as a meta-function by assigning it to the relevant property. Meta-functions are not called in the following cases: x[y]: Using square brackets without a property name only invokes the __Item property. x(): Calling the object itself only invokes the Call method. This includes internal calls made by built-in functions such as SetTimer and Hotkey. Internal calls to other meta-functions or double-underscore methods do not trigger __Call. Dynamic Properties Property syntax and DefineProp can be used to define properties which compute a value each time they are evaluated, but each property must be defined in advance. By contrast, __Get and __Set can be used to implement properties which are known only at the moment they are invoked. For example, a "proxy" object could be created which sends requests for properties over the network (or through some other channel). A remote server would send back a response containing the value of the property, and the proxy would return the value to its caller. Even if the name of each property was known in advance, it would not be logical to define each property individually in the proxy class since every property does the same thing (send a network request). Meta-functions receive the property name as a parameter, so are a good solution to this problem. Primitive Values Primitive values, such as strings and numbers, cannot have their own properties and methods. However, primitive values support the same kind of delegation as objects. That is, any property or method call on a primitive value is delegated to a predefined prototype object, which is also accessible via the Prototype property of the corresponding class. The following classes relate to primitive values: Primitive (extends Any) Number Float Integer String Although checking the Type string is generally faster, the type of a value can be tested by checking whether it has a given base. For example, n.HasBase(Number.Prototype) or n is Number is true if n is a pure Integer or Float, but not if n is a numeric string, since String does not derive from Number. By contrast, IsNumber(n) is true if n is a number or a numeric string. ObjGetBase and the Base property return one of the predefined prototype objects when appropriate. Note that x is Any would ordinarily be true for any value within AutoHotkey's type hierarchy, but false for COM objects. Adding Properties and Methods Properties and methods can be added for all values of a given type by modifying that type's prototype object. However, since a primitive value is not an Object and cannot have its own properties or methods, the primitive prototype objects do not derive from Object.Prototype. In other words, methods such as DefineProp and HasOwnProp are not accessible by default. They can be called indirectly. For example: DefProp := {}.DefineProp DefProp("".base, "Length", { get: StrLen }) MsgBox A_AhkPath.length " == " StrLen(A_AhkPath) Although primitive values can inherit value properties from their prototype, an exception is thrown if the script attempts to set a value property on a primitive value. For example: "".base.test := 1 ; Don't try this at home. MsgBox "".test ; 1 "".test := 2 ; Error: Property is read-only. Although __Set and property setters can be used, they are not useful since primitive values should be considered immutable. Implementation Reference-Counting AutoHotkey uses a basic reference-counting mechanism to automatically free the resources used by an object when it is no longer referenced by the script. Script authors should not invoke this mechanism explicitly, except when dealing directly with unmanaged pointers to objects. Temporary references returned by functions, methods or operators within an expression are released after evaluation of that expression has completed or been aborted. In the following example, the new GMem object is freed only after MsgBox has returned: MsgBox DllCall("GlobalSize", "ptr", GMem(0, 20).ptr, "ptr") ; 20 Note: In this example, .ptr could have been omitted since the Ptr arg type permits objects with a Ptr property. However, the pattern shown above will work even with other property names. To run code when the last reference to an object is being released, implement the __Delete meta-function. Known Limitations: Circular references must be broken before an object can be freed. For details and an example, see Freeing Objects. Although references in static and global variables are released automatically when the program exits, references in non-static local variables or on the expression evaluation stack are not. These references are only released if the function or expression is allowed to complete normally. Although memory used by the object is reclaimed by the operating system when the program exits, __Delete will not be called unless all references to the object are freed. This can be important if it frees other resources which are not automatically reclaimed by the operating system, such as temporary files. Pointers to Objects In some rare cases it may be necessary to pass an object to external code via DllCall or store it in a binary data structure for later retrieval. An object's address can be retrieved via address := ObjPtr(myObject); however, this effectively makes two references to the object, but the program only knows about the one in myObject. If the last known reference to the object was released, the object would be deleted. Therefore, the script must inform the object that it has gained a reference. This can be done as follows (the two lines below are equivalent): ObjAddRef(address := ObjPtr(myObject)) address := ObjPtrAddRef(myObject) The script must also inform the object when it is finished with that reference: ObjRelease(address) Generally each new copy of an object's address should be treated as another reference to the object, so the script should call ObjAddRef when it gains a copy and ObjRelease immediately before losing one. For example, whenever an address is copied via something like x := address, ObjAddRef should be called. Similarly, when the script is finished with x (or is about to overwrite x's value), it should call ObjRelease. To convert an address to a proper reference, use the ObjFromPtr function: myObject := ObjFromPtr(address) ObjFromPtr assumes that address is a counted reference, and claims ownership of it. In other words, myObject := "" would cause the reference originally represented by address to be released. After that, address must be considered invalid. To instead make a new reference, use one of the following: ObjAddRef(address), myObject := ObjFromPtr(address) myObject := ObjFromPtrAddRef(address) Built-in Classes | AutoHotkey v2 Built-in Classes Any Object Array Buffer ClipboardAll Class Error MemoryError OSError TargetError TimeoutError TypeError UnsetError MemberError PropertyError MethodError UnsetItemError ValueError IndexError ZeroDivisionError File Func BoundFunc Closure Enumerator Gui Gui.Control Gui.ActiveX Gui.Button Gui.CheckBox Gui.Custom Gui.DateTime Gui.Edit Gui.GroupBox Gui.Hotkey Gui.Link Gui.List Gui.ComboBox Gui.DDL Gui.ListBox Gui.Tab Gui.ListView Gui.MonthCal Gui.Pic Gui.Progress Gui.Radio Gui.Slider Gui.StatusBar Gui.Text Gui.TreeView Gui.UpDown InputHook Map Menu MenuBar RegExMatchInfo Primitive Number Float Integer String VarRef ComValue ComObjArray ComObject ComValueRef Using the Program | AutoHotkey v2 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 Create a Script Edit a Script Run a Script Tray Icon Main Window Embedded Scripts Command Line Usage 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): 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 FromTitle ExpressionExample .ahk fileA_ScriptFullPath " - AutoHotkey v" A_AhkVersionE:\My Script.ahk - AutoHotkey v1.1.33.09 Main resource (compiled script)A_ScriptFullPathE:\My Script.exe Any other resourceA_ScriptFullPath " - " A_LineFileE:\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: IDSpecUsage 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: SwitchMeaning /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. Scripts - Definition & Usage | AutoHotkey v2 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 Script Startup (the Auto-execute Thread): Taking action immediately upon starting the script, and changing default settings. Splitting a Long Line into a Series of Shorter Ones: This can improve a script's readability and maintainability. Script Library Folders Convert a Script to an EXE (Ahk2Exe): Convert a .ahk script into a .exe file that can run on any PC. Passing Command Line Parameters to a Script: The variable A_Args contains the incoming parameters. Script File Codepage: Using non-ASCII characters safely in scripts. Debugging a Script: How to find the flaws in a misbehaving script. 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 , 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 pair Meaning /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.htmMPRESS 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: SwitchMeaningWorks 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 /Debug Connect 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. Searching... Help Settings Help Settings Specify the settings the help should be started with by default. These settings are permanent; this is, they apply for all current and future visits to this help. Content font size: Very small Small Medium (default) Large Very large Selected tab: Content (default) Index Search Sidebar visibility: Hidden Visible (default) Color theme: Light (default) Dark Quick reference visibility: Uncollapsed (default) Collapsed Note: If you press the following button, ActiveX is used to create a config file named 'chm_config.js' in the same directory as the CHM file, with the following content: Save settings Permission denied. Run the CHM file as administrator or create the config file manually. Beginner Tutorial | AutoHotkey v2 AutoHotkey Beginner Tutorial by tidbit Table of Contents The Basics Downloading and installing AutoHotkey How to create a script How to find the help file on your computer Hotkeys & Hotstrings Keys and their mysterious symbols Window specific hotkeys/hotstrings Multiple hotkeys/hotstrings per file Examples Sending Keystrokes Games Running Programs & Websites Function Calls with or without Parentheses Code blocks Variables Getting user input Other Examples? Objects Creating Objects Using Objects Other Helpful Goodies The mysterious square brackets Finding your AHK version Trial and Error Indentation Asking for Help Other links 1 - The Basics Before we begin our journey, let me give some advice. Throughout this tutorial you will see a lot of text and a lot of code. For optimal learning power, it is advised that you read the text and try the code. Then, study the code. You can copy and paste most examples on this page. If you get confused, try reading the section again. a. Downloading and installing AutoHotkey Since you're viewing this documentation locally, you've probably already installed AutoHotkey and can skip to section b. Before learning to use AutoHotkey (AHK), you will need to download it. After downloading it, you may possibly need to install it. But that depends on the version you want. For this guide we will use the Installer since it is easiest to set up. Text instructions: Go to the AutoHotkey Homepage: https://www.autohotkey.com/ Click Download. You should be presented with an option for each major version of AutoHotkey. This documentation is for v2, so choose that option or switch to the v1 documentation. The downloaded file should be named AutoHotkey_*_setup.exe or similar. Run this and click Install. Once done, great! Continue on to section b. b. How to create a script Once you have AutoHotkey installed, you will probably want it to do stuff. AutoHotkey is not magic, we all wish it was, but it is not. So we will need to tell it what to do. This process is called "Scripting". Text instructions: Right-Click on your desktop. Find "New" in the menu. Click "AutoHotkey Script" inside the "New" menu. Give the script a new name. It must end with a .ahk extension. For example: MyScript.ahk Find the newly created file on your desktop and right-click it. Click "Edit Script". A window should have popped up, probably Notepad. If so, SUCCESS! So now that you have created a script, we need to add stuff into the file. For a list of all built-in function and variables, see section 5. Here is a very basic script containing a hotkey which types text using the Send function when the hotkey is pressed: ^j:: { Send "My First Script" } We will get more in-depth later on. Until then, here's an explanation of the above code: ^j:: is the hotkey. ^ means Ctrl, j is the letter J. Anything to the left of :: are the keys you need to press. Send "My First Script" is how you send keystrokes. Send is the function, anything after the space inside the quotes will be typed. { and } marks the start and end of the hotkey. Save the File. Double-click the file/icon in the desktop to run it. Open notepad or (anything you can type in) and press Ctrl and J. Hip Hip Hooray! Your first script is done. Go get some reward snacks then return to reading the rest of this tutorial. For a video instruction, watch Install and Hello World on YouTube. c. How to find the help file on your computer Downloads for v2.0-a076 and later include an offline help file in the same zip as the main program. If you manually extracted the files, the help file should be wherever you put it. v2.0-beta.4 and later include an installation script. If you have used this to install AutoHotkey, the help file for each version should be in a subdirectory of the location where AutoHotkey was installed, such as "C:\Program Files\AutoHotkey\v2.0-beta.7". There may also be a symbolic link named "v2" pointing to the subdirectory of the last installed version. If v1.x is installed, a help file for that version may also be present in the root directory. Look for AutoHotkey.chm or a file that says AutoHotkey and has a yellow question mark on it. If you don't need to find the file itself, there are also a number of ways to launch it: Via the "Help" menu option in tray menu of a running script. Via the "Help" menu in the main window of a running script, or by pressing F1 while the main window is active. Via the "Help files (F1)" option in the Dash, which can be activated with the mouse or by pressing F1 while the Dash is active. The Dash can be opened via the "AutoHotkey" shortcut in the Start menu. 2 - Hotkeys & Hotstrings What is a hotkey? A hotkey is a key that is hot to the touch. ... Just kidding. It is a key or key combination that the person at the keyboard presses to trigger some actions. For example: ^j:: { Send "My First Script" } What is a hotstring? Hotstrings are mainly used to expand abbreviations as you type them (auto-replace), they can also be used to launch any scripted action. For example: ::ftw::Free the whales The difference between the two examples is that the hotkey will be triggered when you press Ctrl+J while the hotstring will convert your typed "ftw" into "Free the whales". "So, how exactly does a person such as myself create a hotkey?" Good question. A hotkey is created by using a single pair of colons. The key or key combo needs to go on the left of the ::. And the content needs to go below, enclosed in curly brackets. Note: There are exceptions, but those tend to cause confusion a lot of the time. So it won't be covered in the tutorial, at least, not right now. Esc:: { MsgBox "Escape!!!!" } A hotstring has a pair of colons on each side of the text you want to trigger the text replacement. While the text to replace your typed text goes on the right of the second pair of colons. Hotstrings, as mentioned above, can also launch scripted actions. That's fancy talk for "do pretty much anything". Same with hotkeys. ::btw:: { MsgBox "You typed btw." } A nice thing to know is that you can have many lines of code for each hotkey, hotstring, label, and a lot of other things we haven't talked about yet. ^j:: { MsgBox "Wow!" MsgBox "There are" Run "notepad.exe" WinActivate "Untitled - Notepad" WinWaitActive "Untitled - Notepad" Send "7 lines{!}{Enter}" SendInput "inside the CTRL{+} J hotkey." } a. Keys and their mysterious symbols You might be wondering "How the crud am I supposed to know that ^ means Ctrl?!". Well, good question. To help you learn what ^ and other symbols mean, gaze upon this chart: Symbol Description # Win (Windows logo key) ! Alt ^ Ctrl + Shift & An ampersand may be used between any two keys or mouse buttons to combine them into a custom hotkey. (For the full list of symbols, see the Hotkey page) Additionally, for a list of all/most hotkey names that can be used on the left side of a hotkey's double-colon, see List of Keys, Mouse Buttons, and Joystick Controls. You can define a custom combination of two (and only two) keys (except joystick buttons) by using & between them. In the example below, you would hold down Numpad0 then press Numpad1 or Numpad2 to trigger one of the hotkeys: Numpad0 & Numpad1:: { MsgBox "You pressed Numpad1 while holding down Numpad0." } Numpad0 & Numpad2:: { Run "notepad.exe" } But you are now wondering if hotstrings have any cool modifiers since hotkeys do. Yes, they do! Hotstring modifiers go between the first set of colons. For example: :*:ftw::Free the whales Visit Hotkeys and Hotstrings for additional hotkey and hotstring modifiers, information and examples. b. Window specific hotkeys/hotstrings Sometime you might want a hotkey or hotstring to only work (or be disabled) in a certain window. To do this, you will need to use one of the fancy commands with a # in-front of them, namely #HotIf, combined with the built-in function WinActive or WinExist: #HotIf WinActive(WinTitle) #HotIf WinExist(WinTitle) This special command (technically called "directive") creates context-sensitive hotkeys and hotstrings. Simply specify a window title for WinTitle. But in some cases you might want to specify criteria such as HWND, group or class. Those are a bit advanced and are covered more in-depth here: The WinTitle Parameter & the Last Found Window. #HotIf WinActive("Untitled - Notepad") #Space:: { MsgBox "You pressed WIN+SPACE in Notepad." } To turn off context sensitivity for subsequent hotkeys or hotstrings, specify #HotIf without its parameter. For example:; Untitled - Notepad #HotIf WinActive("Untitled - Notepad") !q:: { MsgBox "You pressed ALT+Q in Notepad." } ; Any window that isn't Untitled - Notepad #HotIf !q:: { MsgBox "You pressed ALT+Q in any window." } When #HotIf directives are never used in a script, all hotkeys and hotstrings are enabled for all windows. The #HotIf directive is positional: it affects all hotkeys and hotstrings physically beneath it in the script, until the next #HotIf directive. ; Notepad #HotIf WinActive("ahk_class Notepad") #Space:: { MsgBox "You pressed WIN+SPACE in Notepad." } ::msg::You typed msg in Notepad ; MSPaint #HotIf WinActive("Untitled - Paint") #Space:: { MsgBox "You pressed WIN+SPACE in MSPaint!" } ::msg::You typed msg in MSPaint! For more in-depth information, check out the #HotIf page. c. Multiple hotkeys/hotstrings per file This, for some reason crosses some people's minds. So I'll set it clear: AutoHotkey has the ability to have as many hotkeys and hotstrings in one file as you want. Whether it's 1, or 3253 (or more). #i:: { Run "https://www.google.com/" } ^p:: { Run "notepad.exe" } ~j:: { Send "ack" } :*:acheiv::achiev ::achievment::achievement ::acquaintence::acquaintance :*:adquir::acquir ::aquisition::acquisition :*:agravat::aggravat :*:allign::align : The above code is perfectly acceptable. Multiple hotkeys, multiple hotstrings. All in one big happy script file. d. Examples ::btw::by the way ; Replaces "btw" with "by the way" as soon as you press an default ending character. :*:btw::by the way ; Replaces "btw" with "by the way" without needing an ending character. ^n:: ; CTRL+N hotkey { Run "notepad.exe" ; Run Notepad when you press CTRL+N. } ; This ends the hotkey. The code below this will not be executed when pressing the hotkey. ^b:: ; CTRL+B hotkey { Send "{Ctrl down}c{Ctrl up}" ; Copies the selected text. ^c could be used as well, but this method is more secure. SendInput "[b]{Ctrl down}v{Ctrl up}[/b]" ; Wraps the selected text in BBCode tags to make it bold in a forum. } ; This ends the hotkey. The code below this will not be executed when pressing the hotkey. 3 - Sending Keystrokes So now you decided that you want to send (type) keys to a program. We can do that. Use the Send function. This function literally sends keystrokes, to simulate typing or pressing of keys. But before we get into things, we should talk about some common issues that people have. Just like hotkeys, the Send function has special keys too. Lots and lots of them. Here are the four most common symbols: Symbol Description ! Sends Alt. For example, Send "This is text!a" would send the keys "This is text" and then press Alt+A. Note: !A produces a different effect in some programs than !a. This is because !A presses Alt+Shift+A and !a presses Alt+A. If in doubt, use lowercase. + Sends Shift. For example, Send "+abC" would send the text "AbC", and Send "!+a" would press Alt+Shift+A. ^ Sends Ctrl. For example, Send "^!a" would press Ctrl+Alt+A, and Send "^{Home}" would send Ctrl+Home. Note: ^A produces a different effect in some programs than ^a. This is because ^A presses Ctrl+Shift+A and ^a presses Ctrl+A. If in doubt, use lowercase. # Sends Win (the key with the Windows logo) therefore Send "#e" would hold down Win and then press E. The gigantic table on the Send page shows pretty much every special key built-in to AHK. For example: {Enter} and {Space}. Caution: This table does not apply to hotkeys. Meaning, you do not wrap Ctrl or Enter (or any other key) inside curly brackets when making a hotkey. An example showing what shouldn't be done to a hotkey: ; When making a hotkey... ; WRONG {LCtrl}:: { Send

"AutoHotkey" } ; CORRECT LCtrl:: { Send "AutoHotkey" } A common issue lots of people have is, they assume that the curly brackets are put in the documentation pages just for fun. But in fact they are needed. It's how AHK knows that {!} means "exclamation point" and not "press Alt". So please remember to check the table on the Send page and make sure you have your brackets in the right places. For example: Send "This text has been typed{!}" ; Notice the ! between the curly brackets? That's because if it wasn't, AHK would press the ALT key. ; Same as above, but with the ENTER key. AHK would type out "Enter" if ; it wasn't wrapped in curly brackets. Send "Multiple Enter lines have Enter been sent." ; WRONG Send "Multiple{Enter}lines have{Enter}been sent." ; CORRECT Another common issue is that people think that everything needs to be wrapped in brackets with the Send function. That is FALSE. If it's not in the chart, it does not need brackets. You do not need to wrap common letters, numbers or even some symbols such as . (period) in curly brackets. Also, with the Send functions you are able to send more than one letter, number or symbol at a time. So no need for a bunch of Send functions with one letter each. For example: Send "{a}" ; WRONG Send "{b}" ; WRONG Send "{c}" ; WRONG Send "{a}{b}{c}" ; WRONG Send "{abc}" ; WRONG Send "abc" ; CORRECT To hold down or release a key, enclose the key name in curly brackets and then use the word UP or DOWN. For example: ; This is how you hold one key down and press another key (or keys). ; If one method doesn't work in your program, please try the other. Send "^s" ; Both of these send CTRL+S Send "{Ctrl down}s{Ctrl up}" ; Both of these send CTRL+S Send "{Ctrl down}c{Ctrl up}" Send "{b down}{b up}" Send "{Tab down}{Tab up}" Send "{Up down}" ; Press down the up-arrow key. Sleep 1000 ; Keep it down for one second. Send "{Up up}" ; Release the up-arrow key. But now you are wondering "How can I make my really long Send functions readable?". Easy. Use what is known as a continuation section. Simply specify an opening parenthesis on a new line, then your content, finally a closing parenthesis on its own line. For more information, read about Continuation Sections. Send " (Line 1 Line 2 Apples are a fruit.)" Note: There are several different forms of Send. Each has their own special features. If one form of Send does not work for your needs, try another type of Send. Simply replace the function name "Send" with one of the following: SendText, SendInput, SendPlay, SendEvent. For more information on what each one does, read this. a. Games This is important: A lot of games, especially modern ones, have cheat prevention software. Things like GameGuard, Hackshield, PunkBuster and several others. Not only is bypassing these systems in violation of the games policies and could get you banned, they are complex to work around. If a game has a cheat prevention system and your hotkeys, hotstrings and Send functions do not work, you are out of luck. However there are methods that can increase the chance of working in some games, but there is no magical "make it work in my game now!!!" button. So try ALL of these before giving up. There are also known issues with DirectX. If you are having issues and you know the game uses DirectX, try the stuff described on the FAQ page. More DirectX issues may occur when using PixelSearch, PixelGetColor or ImageSearch. Colors might turn out black (0x000000) no matter the color you try to get. You should also try running the game in windowed mode, if possible. That fixes some DirectX issues. There is no single solution to make AutoHotkey work in all programs. If everything you try fails, it may not be possible to use AutoHotkey for your needs. 4 - Running Programs & Websites To run a program such as mspaint.exe, calc.exe, script.ahk or even a folder, you can use the Run function. It can even be used to open URLs such as https://www.autohotkey.com/ . If your computer is setup to run the type of program you want to run, it's very simple: ; Run a program. Note that most programs will require a FULL file path: Run A_ProgramFiles "\Some_Program\Program.exe" ; Run a website: Run "https://www.autohotkey.com" There are some other advanced features as well, such as command line parameters and CLSID. If you want to learn more about that stuff, visit the Run page. Here are a few more samples: ; Several programs do not need a full path, such as Windows-standard programs: Run "notepad.exe" Run "mspaint.exe" ; Run the "My Documents" folder using a built-in variable: Run A_MyDocuments ; Run some websites: Run "https://www.autohotkey.com" Run "https://www.google.com" For more in-depth information and examples, check out the Run page. 5 - Function Calls with or without Parentheses In AutoHotkey, function calls can be specified with or without parentheses. The parentheses are usually only necessary if the return value of the function is needed or the function name is not written at the start of the line. A list of all built-in functions can be found here. A typical function call looks like this: Function(Parameter1, Parameter2, Parameter3) ; with parentheses Function Parameter1, Parameter2, Parameter3 ; without parentheses The parameters support any kind of expression; this means for example: You can do math in them: SubStr(37 * 12, 1, 2) SubStr(A_Hour - 12, 2) You can call another functions inside them (note that these function calls must be specified with parentheses as they are not at the start of the line): SubStr(A_AhkPath, InStr(A_AhkPath, "AutoHotkey")) Text needs to be wrapped in quotes: SubStr("I'm scripting, awesome!", 16) The most common way assigning the return value of a function to a variable is like so: MyVar := SubStr("I'm scripting, awesome!", 16) This isn't the only way, but the most common. You are using MyVar to store the return value of the function that is to the right of the := operator. See Functions for more details. In short: ; These are function calls without parentheses: MsgBox "This is some text." StrReplace Input, "AutoHotKey", "AutoHotkey" SendInput "This is awesome{!}{!}{!}" ; These are function calls with parentheses: SubStr("I'm scripting, awesome!", 16) FileExist(VariableContainingPath) Output := SubStr ("I'm scripting, awesome!", 16) a. Code blocks Code blocks are lines of code surrounded by little curly brackets ({ and }). They group a section of code together so that AutoHotkey knows it's one big family and that it needs to stay together. They are most often used with functions and control flow statements such as If and Loop. Without them, only the first line in the block is called. In the following code, both lines are run only if MyVar equals 5: if (MyVar = 5) { MsgBox "MyVar equals " MyVar "!!" ExitApp } In the following code, the message box is only shown if MyVar equals 5. The script will always exit, even if MyVar is not 5: if (MyVar = 5) MsgBox "MyVar equals " MyVar "!!" ExitApp This is perfectly fine since the if-statement only had one line of code associated with it. It's exactly the same as above, but I outdented the second line so we know it's separated from the if-statement: if (MyVar = 5) MsgBox "MyVar equals " MyVar "!!" MsgBox "We are now 'outside' of the if-statement. We did not need curly brackets since there was only one line below it." 6 - Variables Variables are like little post-it notes that hold some information. They can be used to store text, numbers, data from functions or even mathematical equations. Without them, programming and scripting would be much more tedious. Variables can be assigned a few ways. We'll cover the most common forms. Please pay attention to the colon equal sign (:=). Text assignment MyVar := "Text" This is the simplest form for a variable. Simply type in your text and done. Any text needs to be in quotes. Variable assignment MyVar := MyVar2 Same as above, but you are assigning a value of a variable to another variable. Number assignment MyVar := 6 + 8 / 3 * 2 - Sqrt(9) Thanks to expressions, you can do math! Mixed assignment MyVar := "The value of 5 + " MyVar2 " is: " 5 + MyVar2 A combination of the three assignments above. Equal signs (=) with a symbol in front of it such as := += -= .= etc. are called assignment operators and always require an expression. a. Getting user input Sometimes you want to have the user to choose the value of stuff. There are several ways of doing this, but the simplest way is InputBox. Here is a simple example on how to ask the user a couple of questions and doing some stuff with what was entered: OutputVar := InputBox("What is your first name?", "Question 1").Value if (OutputVar = "Bill") MsgBox "That's an awesome name, " OutputVar "." OutputVar2 := InputBox("Do you like AutoHotkey?", "Question 2").Value if (OutputVar2 = "yes") MsgBox "Thank you for answering " OutputVar2 ", " OutputVar "! We will become great friends." else MsgBox OutputVar ", That makes me sad." b. Other Examples? Result := MsgBox("Would you like to continue?",, 4) if Result = "No" return ; If No, stop the code from going further. MsgBox "You pressed YES." ; Otherwise, the user picked yes. Var := "text" ; Assign some text to a variable. Num := 6 ; Assign a number to a variable. Var2 := Var ; Assign a variable to another. Var3 .= Var ; Append a variable to the end of another. Var4 += Num ; Add the value of a variable to another. Var4 -= Num ; Subtract the value of a variable from another. Var5 := SubStr(Var, 2, 2) ; Variable inside a function. Var6 := Var "Text" ; Assigns a variable to another with some extra text. MsgBox(Var) ; Variable inside a function. MsgBox Var ; Same as above. Var := StrSplit(Var, "x") ; Variable inside a function that uses InputVar and OutputVar. if (Num = 6) ; Check if a variable is equal to a number. if Num = 6 ; Same as above. if (Var != Num) ; Check if a variable is not equal to another. if Var1 < Var2 ; Check if a variable is lesser than another. 7 - Objects Objects are a way of organizing your data for more efficient usage. An object is basically a collection of variables. A variable that belongs to an object is known as a "property". An object might also contain items, such as array elements. There are a number of reasons you might want to use an object for something. Some examples: You want to have a numbered list of things, such as a grocery list (this would be referred to as an indexed array) You want to represent a grid, perhaps for a board game (this would be done with nested objects) You have a list of things where each thing has a name, such as the characteristics of a fruit (this would be referred to as an associative array) a. Creating Objects There are a few ways to create an object, and the most common ones are listed below: Bracket syntax (Array) MyArray := ["one", "two", "three", 17] This creates an Array, which represents a list of items, numbered 1 and up. In this example, the value "one" is stored at index 1, and the value 17 is stored at index 4. Brace syntax Banana := {Color: "Yellow", Taste: "Delicious", Price: 3} This creates an ad hoc Object. It is a quick way to create an object with a short set of known properties. In this example, the value "Yellow" is stored in the property Color and the value 3 is stored in the property Price. Array constructor MyArray := Array("one", "two", "three", 17) This is equivalent to the bracket syntax. It is actually calling the Array class, not a function. Map constructor MyMap := Map("^", "Ctrl", "!", "Alt") This creates a Map, or associative array. In this example, the value "Ctrl" is associated with the key "^", and the value "Alt" is associated with the key "!". Maps are often created empty with Map() and later filled with items. Other constructor Banana := Fruit() Creates an object of the given class (Fruit in this case). b. Using Objects There are many ways to use objects, including retrieving values, setting values, adding more values, and more. To set values: Bracket notation MyArray[2] := "TWO" MyMap["#"] := "Win" Setting array elements or items in a map or collection is similar to assigning a value to a variable. Simply append bracket notation to the variable which contains the object (array, map or whatever). The index or key between the brackets is an expression, so quote marks must be used for any non-numeric literal value. Dot notation Banana.Consistency := "Mushy" This example assigns a new value to a property of the object contained by Banana. If the property doesn't already exist, it is created. To retrieve values: Bracket notation Value := MyMap["^"] This example retrieves the value previously associated with (mapped to) the key "^". Often the key would be contained by a variable, such as MyMap[modifierChar]. Dot notation Value := Banana.Color This example retrieves the Color property of the object Banana. To add new keys and values: Bracket notation MyMap["NewerKey"] := 3.1415 To directly add a key and value, just set a key that doesn't exist yet. However, note that when assigning to an Array, the index must be within the range of 1 to the array's current length. Different objects may have different requirements. Dot notation MyObject.NewProperty := "Shiny" As mentioned above, assigning to a property that hasn't already been defined will create a new property. InsertAt method MyArray.InsertAt(Index, Value1, Value2, Value3...) InsertAt is a method used to insert new values at a specific position within an

Array, but other kinds of objects may also define a method by this name. Push method MyArray.Push(Value1, Value2, Value3...) Push "appends" the values to the end of the Array MyArray. It is the preferred way to add new elements to an array, since the bracket notation can't be used to assign outside the current range of values. To remove properties and items: Delete method RemovedValue := MyObject.Delete(AnyKey) Array and Map have a Delete method, which removes the value from the array or map. The previous value of MyObject[AnyKey] will be stored in RemovedValue. For an Array, this leaves the array element without a value and doesn't affect other elements in the array. Pop method MyArray.Pop() This Array method removes the last element from an array and returns its value. The array's length is reduced by 1. RemoveAt method RemovedValue := MyArray.RemoveAt(Index) MyArray.RemoveAt(Index, Length) Array has the RemoveAt method, which removes an array element or range of array elements. Elements (if any) to the right of the removed elements are shifted to the left to fill the vacated space. 8 - Other Helpful Goodies We have reached the end of our journey, my good friend. I hope you have learned something. But before we go, here are some other things that I think you should know. Enjoy! a. The mysterious square brackets Throughout the documentation, you will see these two symbols ([and]) surrounding code in the yellow syntax box at the top of almost all pages. Anything inside of these brackets are OPTIONAL. Meaning the stuff inside can be left out if you don't need them. When writing your code, it is very important to NOT type the square brackets in your code. On the ControlGetText page you will see this: Text := ControlGetText(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) So you could simply do this if you wanted: Text := ControlGetText(Control) Or add in some more details: Text := ControlGetText(Control, WinTitle) What if you wanted to use ExcludeTitle but not fill in WinText or WinTitle? Simple! Text := ControlGetText(Control,,, ExcludeTitle) Please note that you cannot IGNORE parameters, but you can leave them blank. If you were to ignore WinTitle, WinText, it would look like this and cause issues: Text := ControlGetText(Control, ExcludeTitle) b. Finding your AHK version Run this code to see your AHK version: MsgBox A_AhkVersion Or look for "AutoHotkey Help File" or "AutoHotkey.chm" in the start menu or your installation directory. c. Trial and Error Trial and Error is a very common and effective way of learning. Instead of asking for help on every little thing, sometimes spending some time alone (sometimes hours or days) and trying to get something to work will help you learn faster. If you try something and it gives you an error, study that error. Then try to fix your code. Then try running it again. If you still get an error, modify your code some more. Keep trying and failing until your code fails no more. You will learn a lot this way by reading the documentation, reading errors and learning what works and what doesn't. Try, fail, try, fail, try, try, try, fail, fail, succeed! This is how a lot of "pros" have learned. But don't be afraid to ask for help, we don't bite (hard). Learning takes time, the "pros" you encounter did not learn to be masters in just a few hours or days. "If at first you don't succeed, try, try, try again." - Hickson, William E. d. Indentation This stuff (indentation) is very important! Your code will run perfectly fine without it, but it will be a major headache for you and other to read your code. Small code (25 lines or less) will probably be fine to read without indentation, but it'll soon get sloppy. It's best you learn to indent ASAP. Indentation has no set style, but it's best to keep everything consistent. "What is indentation?" you ask? It's simply spacing to break up your code so you can see what belongs to what. People usually use 3 or 4 spaces or 1 tab per "level". Not indented: if (car = "old") { MsgBox "The car is really old." if (wheels = "flat") { MsgBox "This car is not safe to drive." return } else { MsgBox "Be careful! This old car will be dangerous to drive." } } else { MsgBox "My, what a shiny new vehicle you have there." } Indented: if (car = "old") { MsgBox "The car is really old." if (wheels = "flat") { MsgBox "This car is not safe to drive." return } else { MsgBox "Be careful! This old car will be dangerous to drive." } } else { MsgBox "My, what a shiny new vehicle you have there." } See Wikipedia's Indentation style page for various styles and examples. Choose what you like or learn to indent how you think it's easiest to read. e. Asking for Help Before you ask, try doing some research yourself or try to code it yourself. If that did not yield results that satisfy you, read below. Don't be afraid to ask for help, even the smartest people ask others for help. Don't be afraid to show what you tried, even if you think it's silly. Post anything you have tried. Pretend everyone but you is a doorknob and knows nothing. Give as much information as you can to educate us doorknobs at what you are trying to do. Help us help you. Be patient. Be polite. Be open. Be kind. Enjoy! If you don't get an answer right away, wait at least 1 day (24 hours) before asking for more help. We love to help, but we also do this for free on our own time. We might be at work, sleeping, gaming, with family or just too busy to help. And while you wait for help, you can try learning and doing it yourself. It's a good feeling, making something yourself without help. f. Other links Frequently Asked Questions (FAQ) Changes from v1.0 to v1.1 | AutoHotkey v2 Changes from v1.0 to v1.1 This document details changes between AutoHotkey v1.0 and v1.1 that may cause scripts to behave differently, and therefore might be important to keep in mind while reading or updating old code. See also: Changes from v1.1 to v2.0. AutoHotkey v1.1 is also known as "AutoHotkey_L", while AutoHotkey v1.0 was retrospectively labelled "AutoHotkey Basic". Some older versions of AutoHotkey_L used 1.0.* version numbers, so for clarity, this document refers to the two branches of AutoHotkey by name rather than version number. Table of Contents High impact: Certain syntax errors are no longer tolerated FileRead may return corrupt binary data Variable and function names do not allow [,] or ? DPI scaling is enabled by default for GUIs Medium impact: Transform's Unicode sub-command is unavailable on Unicode versions AutoHotkey.ahk is launched instead of AutoHotkey.ini SetFormat, Integer, H is case-sensitive A_LastError is modified by more commands MsgBox's handles commas more consistently GUI's Owner option overrides additional styles ~Tilde affects how custom modifier keys work x & y:: causes both x:: and x up:: to fire when x is released Low impact: If var is type ignores the system locale by default GroupActivate sets ErrorLevel and GroupAdd's Label works differently Run and RunWait interpret Target differently Control-Z is not interpreted as end-of-file A_IsCompiled is always read-only DllCall tries the A or W suffix in more cases Syntax Errors Certain syntax errors which were tolerated by AutoHotkey Basic are not tolerated by AutoHotkey_L. Most such errors can be easily corrected once they are identified. Some errors are detected at load-time, and must be corrected before the script will run at all. Other errors are raised only when specific conditions are met while the script is running. Error detection in v2.0 is generally more robust, and as there have been numerous changes to usage beyond just error detection and handling, the differences in error detection between v1.0 and v1.1 are not listed here. For those details, refer to the v1.1 documentation. FileRead FileRead translates text between code pages in certain common cases and therefore might output corrupt binary data. To avoid this in v2.0, add the RAW option or use FileOpen instead. Variable and Function Names The characters [,] and ? are reserved for use in expressions, so are no longer valid in variable names. Consequently, ? (used in ternary operations) no longer requires a space on either side. Code for v1.0 which uses these characters in a variable name would have a new interpretation in v1.1, and as such might not be detected as an error. Related: Operators for Objects, Names (Changes from v1.1 to v2.0) DPI Scaling DPI scaling is enabled by default for script GUIs to ensure they scale according to the system DPI setting. If enabled and the system DPI setting is not 96 (100%), positions and sizes accepted by or returned from Gui methods/properties are not compatible with other functions. To disable DPI scaling, use MyGui.Opt("-DPIScale"). Transform Some Transform sub-commands are altered or unavailable: Transform, Unicode is available only in ANSI versions of AutoHotkey. To assign Unicode text to the clipboard, use a regular assignment. See also: StrPut/StrGet. Transform, HTML supports additional features in Unicode versions of AutoHotkey_L. Transform itself was removed in v2.0. Default Script When AutoHotkey_L is launched without specifying a script, an .ahk file is loaded by default instead of an .ini file. The name of this file depends on the filename of the current executable. For more details, see Script Filename. SetFormat, Integer[Fast], H When an uppercase H is used, hexadecimal digits A-F will also be in uppercase. AutoHotkey Basic always produces lowercase digits. SetFormat itself was removed in v2.0. Format("0x{:x}", n) produces lowercase a-f while Format("0x{:X}", n) produces uppercase A-F. A_LastError The following commands now set A_LastError to assist with debugging: FileAppend, FileRead, FileReadLine, FileDelete, FileCopy, FileMove, FileGetAttrib/Time/Size/Version, FileSetAttrib/Time, FileCreateDir, RegRead, RegWrite, RegDelete. Using any of these commands causes the previous value of A_LastError to be overwritten. For v2.0, A_LastError is also set by IniRead, IniWrite and IniDelete. MsgBox MsgBox in v1.0 and v1.1 had "smart comma handling" to avoid the need to escape commas in unquoted text. This handling was slightly different between the two versions, and might need to be taken into account in very rare cases while reading v1.0 code. Refer to the v1.1 documentation for details. v2.0 uses expression syntax exclusively and as such has no need for any special handling of commas. GUI's Owner option Applying the +Owner option to a Gui also removes the WS_CHILD style and sets the WS_POPUP style. This may break scripts which used +Owner to set the parent window of a Gui after setting the styles. ~Tilde and Custom Combination Hotkeys As of v1.1.14, the tilde prefix affects how a key works when used as a modifier key in a custom combination. Custom Combinations and Down/Up Hotkeys Except when the tilde prefix is used, if both a key-down and a key-up hotkey are defined for a custom modifier key, they will both fire when the key is released. For example, x & y:: causes both x:: and x up:: to fire when x is released, where previously x:: never fired. If var is type If var is type identified certain (possibly locale-specific) non-ASCII characters as alphabetic/uppercase/lowercase by default in v1.0, whereas it did so in v1.1 only if StringCaseSense, Locale was used. Similarly, the IsType functions in v2.0 only identify non-ASCII characters as alphabetic if the second parameter is "Locale". Window Groups GroupActivate sets ErrorLevel only in v1.1, not v1.0 or v2.0. GroupAdd's Label parameter behaves differently between v1.0 and v1.1, but was removed in v2.0. Run / RunWait AutoHotkey_L includes some enhancements to the way the Run and RunWait commands interpret the Target parameter. This allows some things that didn't work previously, but in some very rare cases, may also affect scripts which were already working in AutoHotkey Basic. The new behaviour is as follows: If Target begins with a quotation mark, everything up to the next quotation mark is considered the action, typically an executable file. Otherwise the first substring which ends at a space and is either an existing file or ends in .exe, .bat, .com, .cmd or .hta is considered the action. This allows file types such as .ahk, .vbs or .lnk to accept parameters while still allowing "known" executables such as wordpad.exe to be launched without an absolute path as in previous versions. Control-Z Loop Read and File.ReadLine no longer interpret the character Ctrl+Z (0x1A) as an end-of-file marker. Any Ctrl+Z, even one appearing at the very end of the file, is loaded as-is. FileRead already ignored this character, so is not affected by this issue. A_IsCompiled If the script has not been compiled, A_IsCompiled is defined even if the script has not been compiled; its value is "" in v1.1 and 0 in v2.0. Previously it was left undefined, which meant that assignments such as A_IsCompiled := 1 were valid if the script hadn't been compiled. Now it is treated as a read-only built-in variable in all cases. DllCall When the function name given to DllCall cannot be resolved, AutoHotkey_L automatically appends an "A" (ANSI) or "W" (Unicode) to the function name regardless of which DLL was specified. By contrast, AutoHotkey Basic appends the "A" suffix only for functions in User32.dll, Kernel32.dll, ComCtl32.dll, or Gdi32.dll. Changes from v1.1 | AutoHotkey v2 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 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 ¶m 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() â†' ComObject, which is a class now ComObjParameter() â†' ComValue, which is a class now DriveSpaceFree â†' DriveGetSpaceFree EnvAdd â†' DateAdd EnvSub â†' DateDiff FileCopyDir â†' DirCopy FileCreateDir â†' DirCreate FileMoveDir â†' DirMove FileRemoveDir â†' DirDelete FileSelectFile â†' FileSelect FileSelectFolder â†' DirSelect #If â†' #HotIf #IfTimeout â†' HotIfTimeout StringLower â†' StrLower and StrTitle StringUpper â†' StrUpper and StrTitle UrlDownloadToFile â†' Download WinMenuSelectItem â†' MenuSelect LV, TV and SB functions â†' methods of GuiControl File.__Handle â†' 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 â†' A_LoopFilePath (returns a relative path if the Loop's parameter was relative, so "full path" was misleading) A_LoopFileLongPath â†' A_LoopFileFullPath Clipboard â†' 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 â†' Gui(). Passing an empty title (not omitting it) now results in an empty title, not the default title. Gui Add â†' GuiObj.Add() or GuiObj.AddControlType(); e.g. GuiObj.Add("Edit") or GuiObj.AddEdit(). Gui Show â†' 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 â†’ GuiObj.Submit(). It works like before, except that Submit() creates and returns a new object which contains all of the "associated variables". Gui Destroy â†’ 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 â†’ GuiObj.SetFont(). It is also possible to set a control's font directly, with GuiCtrl.SetFont(). Gui Color â†’ 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 â†’ GuiObj.MarginX and GuiObj.MarginY properties. Gui Menu â†’ GuiObj.MenuBar sets/returns a MenuBar object created with MenuBar(). Gui Cancel/Hide/Minimize/Maximize/Restore â†’ Gui methods of the same name. Gui Flash â†’ GuiObj.Flash(), but use false instead of Off. Gui Tab â†’ 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. ControlDefaultTextNotes ActiveX.Value.TextText is hidden. See below. Button.Text CheckBox.Value.Text ComboBox.TextControlGetText()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.TextUse Value instead of Text if AltSubmit was used. Edit.Value GroupBox.Text Hotkey.Value Link.Text ListBox.TextControlGetText()Use Value instead of Text if AltSubmit was used. Text returns the selected item's text, whereas ControlGetText returns hidden text. See below. ListView.TextText is hidden. MonthCal.Value Picture.Value Progress.Value Radio.Value.Text Slider.Value StatusBar.Text Tab.TextControlGetText()Use Value instead of Text if AltSubmit was used. Text returns the selected tab's text, whereas ControlGetText returns hidden text. Text.Text TreeView.TextText 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 â†’ GuiCtrl.GetPos() Focus â†’ GuiObj.FocusedCtrl; returns a GuiControl object instead of the ClassNN. FocusV â†’ GuiObj.FocusedCtrl.Name Hwnd â†’ GuiCtrl.Hwnd; returns a pure integer, not a hexadecimal string. Enabled/Visible/Name â†’ 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)TextNotes ActiveXN/ACommand 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 ListViewN/ACommand had no effect. MonthCal.Value Picture.Value Progress.ValueUse the += operator instead of the + prefix. Radio.Value.Text Slider.ValueUse the += operator instead of the + prefix. StatusBar.Text or SB.SetText() Tab.Delete/Add/Choose Text.Text TreeViewN/ACommand had no effect. UpDown.ValueUse the += operator instead of the + prefix. Other sub-commands Move â†’ GuiCtrl.Move(x, y, w, h) MoveDraw â†’ GuiCtrl.Move(x, y, w, h), GuiCtrl.Redraw() Focus â†’ 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 â†’ set GuiCtrl.Enabled Hide/Show â†’ set GuiCtrl.Visible Choose â†’ GuiCtrl.Choose(n), where in is a pure integer. The |n or ||n mode is not supported (use ControlChoose instead, if needed). ChooseString â†’ 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 â†’ GuiCtrl.SetFont() +/-Option â†’ 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 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 â†' :2), but ::3::::4 is now 3 â†' ::4 rather than 3:: â†' 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: â†' 6 rather than an error, and ::7`::::8 is now 7: â†' :8 rather than 7:: â†' 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). Variables and Expressions -

Definition & Usage | AutoHotkey v2 Variables and Expressions Table of Contents Variables Expressions Operators in Expressions Built-in Variables Variable Capacity and Memory 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.yx.%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: baseexponent. 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. ISINCONTAINS 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: A_Space, A_Tab Script Properties: command line parameters, A_WorkingDir, A_ScriptDir, A_ScriptName, (...more...) Date and Time: A_YYYY, A_MM, A_DD, A_Hour, A_Min, A_Sec, (...more...) Script Settings: A_IsSuspended, A_ListLines, A_TitleMatchMode, (...more...) User Idle Time: A_TimeIdle, A_TimeIdlePhysical, A_TimeIdleKeyboard, A_TimeIdleMouse Hotkeys, Hotstrings, and Custom Menu Items: A_ThisHotkey, A_EndChar, (...more...) Operating System and User Info: A_OSVersion, A_ScreenWidth, A_ScreenHeight, (...more...) Misc: A_Clipboard, A_Cursor, A_EventInfo, (...more...) Loop: A_Index, (...more...) 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\\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\\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\\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\\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\\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\\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\\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. TrueFalse 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). How to Install AutoHotkey | AutoHotkey v2 How to Install AutoHotkey If you have not already downloaded AutoHotkey, you can get it from one of the following locations: https://www.autohotkey.com/ - multiple options are presented when you click Download. https://www.autohotkey.com/download/ - the main download links are for the current exe installer, but there are also links to the current zip package and archive of older versions. Note: This tutorial is for AutoHotkey v2. The main download has a filename like AutoHotkey_2.0_setup.exe. Run this file to begin installing AutoHotkey. If you are not the administrator of your computer, you may need to select the Current user option. Otherwise, the recommended options are already filled in, so just click Install. For users of v1: AutoHotkey v2 includes a launcher which allows multiple versions of AutoHotkey to co-exist while sharing one file extension (.ahk). Installing AutoHotkey v1 and v2 into different directories is not necessary and is currently not supported. If you are installing for all users, you will need to provide administrator consent in the standard UAC prompt that appears (in other words, click Yes). If there were no complications, AutoHotkey is now installed! Once installation completes, the Dash is shown automatically. Next Steps Using the Program covers the basics of how to use AutoHotkey. Consider installing an editor with AutoHotkey support to make editing and testing scripts much easier. The documentation contains many examples, which you can test out as described in How to Run Example Code. These tutorials focus on specific common tasks: How to Run Programs How to Write Hotkeys How to Send Keystrokes AutoHotkey Beginner Tutorial by tidbit covers a range of topics. Problems? If you have problems installing AutoHotkey, please try searching for your issue on the forum or start a new topic to get help or report the issue. Zip Installer AutoHotkey can also be installed from the zip download. Open the zip file using File Explorer (no extraction necessary), or extract the contents of the zip file to a temporary directory. Run Install.cmd. If you receive a prompt like "The publisher could not be verified. Are you sure...?", click Run. Continue installation as described above. Security Prompts You may receive one or more security prompts, depending on several factors. Web Browsers Common web browsers may show a warning like "AutoHotkey_2.0_setup.exe was blocked because it could harm your device." This is a generic warning that applies to any executable file type that isn't "commonly downloaded". In other words, it often happens for new releases of software, until more users have downloaded that particular version. To keep the download, methods vary between browsers. Look for a menu button near where downloads are shown, or try right clicking on the blocked download. Sometimes the download might be blocked due to an antivirus false-positive; in that case, see Antivirus below. The Google Safe Browsing service (also used by other browsers) has been known to show false warnings about

AutoHotkey. For details, see Safe Browsing. SmartScreen Microsoft Defender SmartScreen may show a prompt like "Windows protected your PC". This is common for software from open source developers and Independent Software Vendors (ISV), especially soon after the release of each new version. The following blog article by Louis Kessler describes the problem well: Thatâ™s not very Smart of you, Microsoft To continue installation, select More info and then Run anyway. Antivirus If your antivirus flags the download as malicious, please refer to the following: FAQ: My antivirus program flagged AutoHotkey or a compiled script as malware. Is it really a virus? Report False-Positives To Anti-Virus Companies Antivirus False Positives How to Manage Windows | AutoHotkey v2 How to Manage Windows One of the easiest and most useful things AutoHotkey can do is allow you to create keyboard shortcuts (hotkeys) that manipulate windows. A script can activate, close, minimize, maximize, restore, hide, show or move almost any window. This is done by calling the appropriate Win function, specifying the window by title or some other criteria: Run "notepad.exe" WinWait "Untitled - Notepad" WinActivate "Untitled - Notepad" WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2, "Untitled - Notepad" This example should open a new Notepad window and then move it to fill a portion of the primary screen (¼ of its width and ½ its height). To learn how to try it out, refer to How to Run Example Code. We won't go into detail about many of the functions for manipulating windows, since there's not much to it. For instance, to minimize a window instead of activating it, replace WinActivate with WinMinimize. See Win functions for a list of functions which can manipulate windows or retrieve information. Most of this tutorial is about ways to identify which window you want to operate on, since that is often the most troublesome part. For instance, there are a number of problems with the example above: The title is repeated unnecessarily. The title is correct only for systems with UI language set to English. It might move an existing untitled Notepad window instead of the new one. If for some reason no matching window appears, the script will stall indefinitely. We'll address these issues one at a time, after covering a few basics. Tip: AutoHotkey comes with a script called Window Spy, which can be used to confirm the title, class and process name of a window. The class and process name are often used when identifying a window by title alone is not feasible. You can find Window Spy in the script's tray menu or the AutoHotkey Dash. Title Matching There are a few things to know when specifying a window by title: Window titles are always case-sensitive, except when using the RegEx matching mode with the i) modifier. By default, functions expect a substring of the window title. For example, "Notepad" could match both "Untitled - Notepad" and "C:\A\B.ahk - Notepad++". SetTitleMatchMode can be used to make functions expect a prefix, exact match, or RegEx pattern instead. By default, hidden windows are ignored (except by WinShow). DetectHiddenWindows can be used to change this. See Matching Behaviour for more details. Active Window To refer to the active window, use the letter "A" in place of a window title. For example, this minimizes the active window: WinMinimize "A" Last Found Window When WinWait, WinExist, WinActive, WinWaitActive or WinWaitNotActive find a matching window, they set it as the last found window. Most window functions allow the window title (and related parameters) to be omitted, and will in that case default to the last found window. For example: Run "notepad.exe" WinWait "Untitled - Notepad" WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 This saves repeating the window title, which saves a little of your time, makes the script easier to update if the window title needs to be changed, and might make the code easier to read. It makes the script more reliable by ensuring that it operates on the same window each time, even when there are multiple matching windows, or if the window title changes after the window is "found". It also makes the script execute more efficiently, but not by much. Window Class A window class is a set of attributes that is used as a template to create a window. Often the name of a window's class is related to the app or the purpose of the window. A window's class never changes while the window exists, so we can often use it to identify a window when identifying by title is impractical or impossible. For example, instead of the window title "Untitled - Notepad", we can use the window's class, which in this case happens to be "Notepad" regardless of the system language: Run "notepad.exe" WinWait "ahk_class Notepad" WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 A window class is distinguished from a title by using the word "ahk_class" as shown above. To combine multiple criteria, list the window title first. For example: "Untitled ahk_class Notepad". Related: ahk_class Process Name/Path Windows can be identified by the process which created them by using the word "ahk_exe" followed by the name or path of the process. For example: Run "notepad.exe" WinWait "ahk_exe notepad.exe" WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 Related: ahk_exe Process ID (PID) Each process has an ID number which remains unique until the process exits. We can use this to make our Notepad example more reliable by ensuring that it ignores any Notepad window except the one that is created by the new process: Run "notepad.exe",,, ¬epad_pid WinWait "ahk_pid " notepad_pid WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 We need three commas in a row; two of them are just to skip the unused WorkingDir and Options parameters of the Run function, since the one we want (OutputVarPID) is the fourth parameter. Ampersand (&) is the reference operator. This is used to pass the notepad_pid variable to the Run function by reference (in other words, to pass the variable itself instead of its value), allowing the function to assign a new value to the variable. Then notepad_pid becomes a placeholder for the actual process ID. The string "ahk_pid " is concatenated with the process ID contained by the notepad_pid variable by simply writing them in sequence, separated by whitespace. The result is a string like "ahk_pid 1040", but the number isn't predictable. If the new process might create multiple windows, a window title and other criteria can be combined by delimiting them with spaces. The window title must always come first. For example: "Untitled ahk_class Notepad ahk_pid " notepad_pid. Related: ahk_pid Window ID (HWND) Each window has an ID number which remains unique until the window is destroyed. In programming parlance, this is known as a "window handle" or HWND. Although not as convenient as using the last found window, the window's ID can be stored in a variable so that the script can refer to it by a name of your choice, even if the title changes. There can be only one last found window at a time, but you can use as many window IDs as you can make up variable names for (or you can use an array). A window ID is returned by WinWait, WinExist or WinActive, or can come from some other sources. The Notepad example can be rewritten to take advantage of this: Run "notepad.exe" notepad_id := WinWait("Untitled - Notepad") WinActivate notepad_id WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2, notepad_id This assigns the return value of WinWait to the variable "notepad_id". In other words, when WinWait finds the window, it produces the window's ID as its result, and the script then stores this result in the variable. "notepad_id" is just a name that I've made up for this example; you can use whatever variable names make sense to you (within certain constraints). Notice that I added parentheses around the window title, immediately following the function name. Parentheses can be omitted in function call statements (that is, function calls at the very beginning of the line), but in that case you cannot get the function's return value. The script can also retain the variable notepad_id for later use, such as to close or reactivate the window or move it somewhere else. Related: ahk_id Timeout By default, WinWait will wait indefinitely for a matching window to appear. You can determine whether this has happened by opening the script's main window via the tray icon (unless you've disabled it). The window normally opens on ListLines view by default. If WinWait is still waiting, it will appear at the very bottom of the list of lines, followed by the number of seconds since it began waiting. The number doesn't update unless you select "Refresh" from the View menu. Try running this example and opening the main window as described above: WinWait "Untitled - Notpad" ; (intentional typo) If the script is stuck waiting for a window, you will usually need to exit or reload the script to get it unstuck. To prevent that from happening in the first place (or happening again), you can use the Timeout parameter of WinWait. For example, this will wait at most 5 seconds for the window to appear: Run "notepad.exe",,, ¬epad_pid if WinWait("ahk_pid " notepad_pid,, 5) { WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 } The block below the if statement is executed only if WinWait finds a matching window. If it times out, the block is skipped and execution continues after the closing brace (if there is any code after it). Note that the parentheses after "WinWait" are required when we want to use the function's result in an expression (such as the condition of an if statement). You can think of the function call itself as a substitute for the result of the function. For instance, if WinWait finds a match before timing out, the result is non-zero. if 1 would execute the block below the if statement, whereas if 0 would skip it. Another way to write it is to return early (in other words, abort) if the wait times out. For example: Run "notepad.exe",,, ¬epad_pid if !WinWait("ahk_pid " notepad_pid,, 5) return WinActivate WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2 The result is inverted by applying the logical-not operator (! or not). If WinWait times out, its result is 0. The result of !0 is 1, so when WinWait times out, the if statement executes the return. WinWait's result is actually the ID of the window (as described above) or zero if it timed out. If you also want to refer to the window by ID, you can assign the result to a variable instead of using it directly in the if statement: Run "notepad.exe",,, ¬epad_pid notepad_id := WinWait("ahk_pid " notepad_pid,, 5) if notepad_id { WinActivate notepad_id WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2, notepad_id } To avoid repeating the variable name, you can both assign the result to a variable and check that it is non-zero (true) at the same time: Run "notepad.exe",,, ¬epad_pid if notepad_id := WinWait("ahk_pid " notepad_pid,, 2) { WinActivate notepad_id WinMove 0, 0, A_ScreenWidth/4, A_ScreenHeight/2, notepad_id } In that case, be careful not to confuse := (assignment) with = or == (comparison). For example, if myvar := 0 assigns a new value and gives the same result every time (false), whereas if myvar = 0 compares a previously-assigned value with 0. Expressions (Math etc.) When you want to move a window, it is often useful to move or size it relative to its previous position or size, which can be retrieved by using the WinGetPos function. For example, the following set of hotkeys can be used to move the active window by 10 pixels in each direction, by holding RCtrl and pressing the arrow keys: >^Left:: MoveActiveWindowBy(-10, 0) >^Right:: MoveActiveWindowBy(+10, 0) >^Up:: MoveActiveWindowBy(0, -10) >^Down:: MoveActiveWindowBy(0, +10) MoveActiveWindowBy(x, y) { WinExist "A" ; Make the active window the Last Found Window WinGetPos ¤t_x, ¤t_y WinMove current_x + x, current_y + y } The example defines a function to avoid repeating code several times. x and y become placeholders for the two numbers specified in each hotkey. WinGetPos stores the current position in current_x and current_y, which we then add to x and y. Simple expressions such as this should look fairly familiar. For more details, see Expressions; but be aware there is a lot of detail that you probably won't need to learn immediately. How to Run Example Code | AutoHotkey v2 How to Run Example Code The easiest way to get started quickly with AutoHotkey is to take example code, try it out and adapt it to your needs. Within this documentation, there are many examples in code blocks such as the one below. MsgBox "Hello, world!" Most (but not all) examples can be executed as-is to demonstrate their effect. There are generally two ways to do this: Download the code as a file. If your browser supports it, you can download any code block (such as the one above) as a script file by clicking the â†“ button which appears in the top-right of the code block when you hover your mouse over it. Copy the code into a file. It's usually best to create a new file, so existing code won't interfere with the example code. Once the file has been created, open it for editing and copy-paste the code. Run the file: Once you have the code in a script (.ahk) file, running it is usually just a case of double-clicking the file; but there are other methods. Assigning Hotkeys Sometimes

testing code is easier if you assign it to a hotkey first. For example, consider this code for maximizing the active window: WinMaximize "A" If you save this into a file and run the file by double-clicking it, it will likely maximize the File Explorer window which contains the file. You can instead assign it to a hotkey to test its effect on whatever window you want. For example: ^1::WinMaximize "A" Now you can activate your test subject and press Ctrl+1 to maximize it. For more about hotkeys, see How to Write Hotkeys. Bailing Out If you make a mistake in a script, sometimes it can make the computer harder to use. For example, the hotkey n:: would activate whenever you press N and would prevent you from typing that character. To undo this, all you need to do is exit the script. You can do that by right clicking on the script's tray icon and selecting Exit. Keys can get "stuck down" if you send a key-down and don't send a key-up. In that case, exiting the script won't necessarily be enough, as the operating system still thinks the key is being held down. Generally you can "unstick" the key by physically pressing it. If a script gets into a runaway loop or is otherwise difficult to stop, you can log off or shut down the computer as a last resort. When you log off, all apps running under your session are terminated, including AutoHotkey. In some cases you might need to click "log off anyway" or "shut down anyway" if a script or program is preventing shutdown. Reloading After you've started the script, changes to the script's file do not take effect automatically. In order to make them take effect, you must reload the script. This can be done via the script's tray icon or the Reload function, which you can call from a hotkey. In many cases it can also be done by simply running the script again, but that depends on the script's #SingleInstance setting. The Right Tools Learning to code is often a repetitious process; take some code, make a small change, test the code, rinse and repeat. This process is quicker and more productive if you use a text editor with support for AutoHotkey. Support varies between editors, but the most important features are (in my opinion): The ability to run the script with a keyboard shortcut (such as F5). Syntax highlighting to make the code easier to read (and write). For recommendations, try Editors with AutoHotkey Support or the Editors subforum. How to Run Programs | AutoHotkey v2 How to Run Programs One of the easiest and most useful things AutoHotkey can do is allow you to create keyboard shortcuts (hotkeys) that launch programs. Programs are launched by calling the Run function, passing the command line of the program as a parameter: Run "C:\Windows\notepad.exe" This example launches Notepad. To learn how to try it out, refer to How to Run Example Code. At this stage, we haven't defined a hotkey (in other words, assigned a keyboard shortcut), so the instructions are carried out immediately. In this case, the script has nothing else to do, so it automatically exits. If you prefer to make useful hotkeys while learning, check out How to Write Hotkeys first. Note: Run can also be used to open documents, folders and URLs. To launch other programs, simply replace the path in the example above with the path of the program you wish to launch. Some programs have their paths registered with the system, in which case you can get away with just passing the filename of the program, with or (sometimes) without the ".exe" extension. For example: Run "notepad" Command-line Parameters If the program accepts command-line parameters, they can be passed as part of the Run function's first parameter. The following example should open license.txt in Notepad: Run "notepad C:\Program Files\AutoHotkey\license.txt" Note: This example assumes AutoHotkey is installed in the default location, and will show an error otherwise. Simple, right? Now suppose that we want to open the file in WordPad instead of Notepad. Run "wordpad C:\Program Files\AutoHotkey\license.txt" Run this code and see what you can learn from the result. Okay, so the new code doesn't work. Hopefully you didn't dismiss the error dialog immediately; error dialogs are a normal part of the process of coding, and often contain very useful information. This one should tell us a few things: Firstly, the obvious: the program couldn't be launched. The dialog shows "Action" and "Params", but our whole command line is shown next to "Action", and "Params" is empty. In other words, the Run function doesn't know where the program name ends and the parameters begin. "The system cannot find the file specified" (on English-language systems). Perhaps the system couldn't find "wordpad", but what it's really saying is that there is no such file as "wordpad C:\...". But why did Notepad work? Running either "notepad" or "wordpad" on its own works, but for different reasons. Unlike notepad.exe, wordpad.exe cannot be found by checking each directory listed in the PATH environment variable. It can be located by a different method, which requires the Run function to separate the program name and parameters. So in this case, the Run function needs a bit of help, in any or all of the following forms: Explicitly use the ".exe" extension. Explicitly use the full path of wordpad.exe. Enclose the program name in quotation marks. For now, go with the easiest option: Run "wordpad.exe C:\Program Files\AutoHotkey\license.txt" Now WordPad launches, but it shows an error: "C:\Program" wasn't found. Quote Marks and Spaces Often when passing command-line parameters to a program, it is necessary to enclose each parameter in quote marks if the parameter contains a space. This wasn't necessary with Notepad, but Notepad is an exception to the general rule. A naive attempt at a solution might be to simply add more quote marks: Run "wordpad.exe "C:\Program Files\AutoHotkey\license.txt"" But this won't work, because by default, quote marks are used to denote the start and end of literal text. So how do we include a literal quote mark within the command line, rather than having it end the command line? Method 1: Precede each literal quote mark with ` (back-tick, back-quote or grave accent). This is known as an escape sequence. The quote mark is then included in the command line (i.e. the string that is passed to the Run function), whereas the back-tick, having fulfilled its purpose, is left out. Run "wordpad.exe `"C:\Program Files\AutoHotkey\license.txt`"" Method 2: Enclose the command line in single quotes instead of double quotes. Run 'wordpad.exe "C:\Program Files\AutoHotkey\license.txt"' Of course, in that case any literal single quotes (or apostrophes) in the text would need to be escaped (`'). How you write the code affects which quote marks actually make it through to the Run function. In the two examples above, the Run function receives the string wordpad.exe "C:\Program Files\AutoHotkey\license.txt". The Run function either splits this into a program name and parameters (everything else) or leaves that up to the system. In either case, how the remaining quote marks are interpreted depends entirely on the target program. Many programs treat a quote mark as part of the parameter if it is preceded by a backslash. For example, Run 'my.exe "A\" B' might produce a parameter with the value A" B instead of two parameters. This is up to the program, and can usually be avoided by doubling the backslash, as in Run 'my.exe "A\\" B', which usually produces two parameters (A\ and B). Most programs interpret quote marks as a sort of toggle, switching modes between "space ends the parameter" and "space is included in the parameter". In other words, Run 'my.exe "A B"' is generally equivalent to Run 'my.exe A" "B'. So another way to avoid issues with slashes is to quote the spaces instead of the entire parameter, or end the quote before the slash, as in Run 'my.exe "A"\ B'. Including Variables Often a command line needs to include some variables. For example, the location of the "Program Files" directory can vary between systems, and a script can take this into account by using the A_ProgramFiles variable. If the variable contains the entire command line, simply pass the variable to the Run function to execute it. Run A_ComSpec ; Start a command prompt (almost always cmd.exe). Run A_MyDocuments ; Open the user's Documents folder. Including a variable inside a quoted string won't work; instead, we use concatenation to join literal strings together with variables. For example: Run 'notepad.exe "' A_MyDocuments '\AutoHotkey.ahk"' Another method is to use Format to perform substitution. For example: Run Format('notepad.exe "{1}\AutoHotkey.ahk"', A_MyDocuments) Note: Format can perform additional formatting at the same time, such as padding with 0s or spaces, or formatting numbers as hexadecimal instead of decimal. Run's Parameters Aside from the command line to execute, the Run function accepts a few other parameters that affect its behaviour. WorkingDir specifies the working directory for the new process. If you specify a relative path for the program, it is relative to this directory. Relative paths in command line parameters are often also relative to this directory, but that depends on the program. Run "cmd", "C:\" ; Open a command prompt at C:\ Options can often be used to run a program minimized or hidden, instead of having it pop up on screen, but some programs ignore it. OutputVarPID gives you the process ID, which is often used with WinWait or WinWaitActive and ahk_pid to wait until the program shows a window on screen, or to identify one of its windows. For example: Run "notepad",,, &pid WinWaitActive "ahk_pid " pid SendText "Hello, world!" System Verbs System verbs are actions that the system or applications register for specific file types. They are normally available in the file's right-click menu in Explorer, but their actual names don't always match the text displayed in the menu. For example, AutoHotkey scripts have an "edit" verb which opens the script in an editor, and (if Ahk2Exe is installed) a "compile" verb which compiles the script. "Edit" is one of a list of common verbs that Run recognizes by default, so can be used by just writing the word followed by a space and the filename, as follows: Run 'edit ' A_ScriptFullPath ; Generally equivalent to Edit Any verb registered with the system can be executed by using the * prefix as shown below: Run '*Compile-Gui ' A_ScriptFullPath If Ahk2Exe is installed, this opens the Ahk2Exe Gui with the current script pre-selected. Environment Whenever a new process starts, it generally inherits the environment of the process which launched it (the parent process). This basically means that all of the script's environment variables are inherited by any program that you launch with Run. In some cases, environment variables can be set with EnvSet before launching the program to affect its behaviour, or pass information to it. A script can also use EnvGet to read environment variables that it might have inherited from its parent process. On 64-bit systems, the script's own environment heavily depends on whether the EXE running it is 32-bit or 64-bit. 32-bit processes not only have different environment variables, but also have file system redirection in place for compatibility reasons. Run "cmd /k set pro" The example above shows a command prompt which prints all environment variables beginning with "pro". If you run it from a 32-bit script, you will likely see PROCESSOR_ARCHITECTURE=x86 and ProgramFiles=C:\Program Files (x86). Although the title shows something like "C:\Windows\System32\cmd.exe", this is a lie; it is actually the 32-bit version, which really resides in "C:\Windows\SysWow64\cmd.exe". In simple cases like this, the easiest way to bypass the redirection of "System32" is to use "SysNative" instead. However, this only works from a 32-bit process on a 64-bit system, so must be done conditionally. When the following example is executed on a 64-bit system, it shows a 64-bit command prompt even if the script is 32-bit: if FileExist(A_WinDir "\SysNative") Run A_WinDir "\SysNative\cmd.exe /k set pro" else Run "cmd /k set pro" How to Send Keystrokes | AutoHotkey v2 How to Send Keystrokes Send "Hello, world{!}{Left}^+{Left}" Sending keystrokes (or keys for short) is the most common method of automating programs, because it is the one that works most generally. More direct methods tend to work only in particular types of app. There are broadly two parts to learning how to send keys: How to write the code so that the program knows which keys you want to send. How to use the available modes and options to get the right end result. It is important to understand that sending a key does not perfectly replicate the act of physically pressing the key, even if you slow it down to human speeds. But before we go into that, we'll cover some basics. Trying the examples If you run an example like SendText "Hi!", the text will be immediately sent to the active (focused) window, which might be less than useful depending on how you ran the example. It's usually better to define a hotkey, run the example to load it up, and press the hotkey when you want to test its effect. Some of the examples below will use numbered hotkeys like ^1:: (Ctrl and a number, so you can try multiple examples at once if there are no duplicates), but you can change that to whatever suits you. To learn how to customize the hotkeys or create

your own, see How to Write Hotkeys. If you're not sure how to try out the examples, see How to Run Example Code. How to write the code When sending keys, you generally want to either send a key or key combination for its effect (like Ctrl+C to copy to the clipboard), or type some text. Typing text is simpler, so we'll start there: just call the SendText function, passing it the exact text you want to send. ^1::SendText "To Whom It May Concern" Technically SendText actually sends Unicode character packets and not keystrokes, and that makes it much more reliable for characters that are normally typed with a key-combination like Shift+2 or AltGr+a. Rules of quoted strings SendText sends the text verbatim, but keep in mind the rules of the language. For instance, literal text must be enclosed in quote marks (double " or single '), and the quote marks themselves aren't "seen" by the SendText function. To send a literal quote mark, you can enclose it in the opposite type of quote mark. For example: ^2::SendText 'Quote marks are also known as "quotes".' Alternatively, use an escape sequence. Inside a quoted string, `" translates to a literal " and `' translates to a literal '. For example: ^3::{ SendText "Double quote (`")" SendText 'Single quote (`')' } You can also alternate the quote marks: ^4::SendText 'Double (") and' . " single (') quote" The two strings are joined together (concatenated) before being passed to the SendText function. The dot (.) can be omitted, but that makes it harder to see where one ends and the other begins. As you've seen above, the escape character ` (known by backquote, backtick, grave accent and other names) has special meaning, so if you want to send that character literally (or send the corresponding key), you need to double it up, as in Send "``". Other common escape sequences include `n for linefeed (Enter) and `t for tab. See Escape Sequences for more. Sending keys and key combinations SendText is best for sending text verbatim, but it can't send keys that don't produce text, like Left or Home. Send, SendInput, SendPlay, SendEvent and ControlSend can send both text and key combinations, or keys which don't produce text. To do all of this, they add special meaning to the following symbols: ^!+#{} The first four symbols correspond to the standard modifier keys, Ctrl (^), Alt (!), Shift (+) and Win (#). They can be used in combination, but otherwise affect only the next key. To send a key by name, or to send any one of the above symbols literally, enclose it in braces. For example: ^+{Left} produces Ctrl+Shift+Left ^{+}{Left} produces Ctrl++ followed by Left ^+Left produces Ctrl+Shift+L followed literally by the letters eft When you press Ctrl+Shift+", the following example sends two quote marks and then moves the insertion point to the left, ready to type inside the quote marks: ^+"::Send '""{Left}' For any single character other than ^!+#{}, Send translates it to the corresponding key combination and presses and releases that combination. For example, Send "aB" presses and releases A and then presses and releases Shift+B. Similarly, any key name enclosed in braces is pressed and released by default. For example, Send "{Ctrl}a" would press and release Ctrl, then press and release A; probably not what you want. To only press (hold down) or release a key, enclose the key name in braces, followed by a space and then the word "down" or "up". The following example causes Ctrl+CapsLock to act as a toggle for Shift: *^CapsLock::{ if GetKeyState("Shift") Send "{Shift up}" else Send "{Shift down}" } Hotkeys vs. Send Warning: Hotkeys and Send have some differences that you should be aware of. Although hotkeys also use the symbols ^!+# and the same key names, there are several important differences: Other hotkey modifier symbols are not supported by Send. For example, >^a:: corresponds to RCtrl+A, but to send that combination, you need to spell out the key name in full, as in Send "{RCtrl down}a{RCtrl up}". Key names are never enclosed in braces within hotkeys, but must always be enclosed in braces for Send (if longer than one character). Send is case-sensitive. For example, Send "^A" sends the combination of Ctrl with upper-case "a", so Ctrl+Shift+A. By contrast, ^a:: and ^A:: are equivalent. This is because Send serves multiple purposes, whereas hotkeys are optimized for key combinations. On a related note, hotstrings are exclusively for detecting text entry, so the symbols ^!+#{} have no special meaning within the hotstring trigger text. However, a hotstring's replacement text uses the same syntax as Send (except when the T option is used). Whenever you type "{" with the following hotstring active, it sends "}" and then Left to move the insertion point back between the braces: :*?B0:{::{}}{Left} Blind mode Normally, Send assumes that any modifier keys you are physically holding down should not be combined with the keys you are asking it to send. For instance, if you are holding Ctrl and you call Send "Hi", Send will automatically release Ctrl before sending "Hi" and press it back down afterward. Sometimes what you want is to send some keys in combination with other modifiers that were previously pressed or sent. To do this, you can use the {Blind} prefix. While running the following example, try focusing a non-empty text editor or input field and pressing 1 or 2 while holding Ctrl or Ctrl+Shift: *^1::Send "{Blind}{Home}" *^2::Send "{Blind}{End}" For more about {Blind}, see Blind mode. Others Send supports a few other special constructs, such as: {U+00B5} to send a Unicode character by its ordinal value (character code). {ASC 0181} to send an Alt+Numpad sequence. {Click Options} to click or move the mouse. For a full list, see Key names. Modes and options Sending a key does not perfectly replicate the act of physically pressing the key. The operating system provides several different ways to send keys, with different caveats for each. Sometimes to get the result you want, you will need to not only try different methods but also tweak the timing. The main methods are SendInput, SendEvent and SendPlay. SendInput is generally the most reliable, so by default, Send is synonymous with SendInput. SendMode can be used to make Send synonymous with SendEvent or SendPlay instead. The documentation describes other pros and cons of SendInput and SendPlay at length, but I would suggest just trying SendEvent or SendPlay when you have issues with SendInput. Warning: SendPlay doesn't tend to work on modern systems unless you run with UI access. Another option worth trying is ControlSend. This doesn't use an official method of sending keystrokes, but instead sends messages directly to the window that you specify. The main advantage is that the window usually doesn't need to be active to receive these messages. But since it bypasses the normal processing of keyboard input by the system, sometimes it doesn't work. Timing and delays Sometimes you can get away with sending a flood of keystrokes faster than humanly possible, and sometimes you can't. There are generally two situations where you might need a delay: A keypress is supposed to trigger some change within the target app (such as showing a new control or window), and sending another keypress before that happens will have the wrong effect. The app can't keep up with a rapid stream of keystrokes, and you need to slow them all down. For the first case, you can simply call Send, then Sleep, then Send, and so on. SetKeyDelay exists for the second case. This function can set a delay to be performed between each keystroke, and the duration of the keystroke (i.e. the delay between pressing and releasing the key). ^1::{ SetKeyDelay 75, 25 ; 75ms between keys, 25ms between down/up. SendEvent "You should see the keys{bs 4}text appear gradually." } Warning: SendInput does not support key delays, nor does Send by default. In order to make SetKeyDelay effective, you must generally either use SendMode "Event" or call SendEvent, SendPlay or ControlSend instead of Send or SendText. Sending a lot of text One way to send multiple lines of text is to use a continuation section: SendText " (Leading indentation is stripped out, based on the first line. Line breaks are kept unless you use the "Join" option.)" Although it is generally quite fast, SendText still has to send each character one at a time, while Send generally needs to send at least twice as many messages (key-down and key-up). This adds up to a noticeable delay when sending a large amount of text. It can also become unreliable, as a longer delay means higher risk of conflict with input by the user, the keyboard focus shifting, or other conditions changing. Generally it is faster and more reliable to instead place the text on the clipboard and paste it. For example: ^1::{ old_clip := ClipboardAll() ; Save all clipboard content A_Clipboard := " (Join`s This text is placed on the clipboard, and will be pasted below by sending Ctrl+V.)" Send "^v" Sleep 500 ; Wait a bit for Ctrl+V to be processed A_Clipboard := old_clip ; Restore previous clipboard content } How to Write Hotkeys | AutoHotkey v2 How to Write Hotkeys A hotkey is a key or key combination that triggers an action. For example, Win+E normally launches File Explorer, and F1 often activates an app-specific help function. AutoHotkey has the power to define hotkeys that can be used anywhere or only within specific apps, performing any action that you are able to express with code. The most common way to define a hotkey is to write the hotkey name followed by double-colon, then the action: #n::Run "notepad" This example defines a hotkey that runs Notepad whenever you press Win+N. To learn how to try it out, refer to How to Run Example Code. For more about running programs, see How to Run Programs. If multiple lines are required, use braces to mark the start and end of the hotkey's action. This is called a block. #n:: { if WinExist("ahk_class Notepad") WinActivate ; Activate the window found above else Run "notepad" ; Open a new Notepad window } The opening brace can also be written on the same line as the hotkey, after ::. The block following a double-colon hotkey is implicitly the body of a function, but that is only important when you define your own variables. For now, just know that blocks are used to group multiple lines as a single action or statement (see Control Flow if you want to know more about this). Basic Hotkeys For most hotkeys, the hotkey name consists of optional modifier symbols immediately followed by a single letter or symbol, or a key name. Try making the following changes to the example above: Remove # to make the hotkey activate whenever you press N on its own. Keep in mind that if something goes wrong, you can always exit the script. Replace # with ^ for Ctrl, ! for Alt or + for Shift, or try combinations. Replace # with <^ to make it activate only when the left Ctrl key is pressed, or >^ for the right Ctrl key, or both to require both keys. Replace n with any other letter or symbol (except :). Replace n with any name from the key list. Note: The last character before :: is never interpreted as a modifier symbol. With this form of hotkey, only the final key in the combination can be written literally as a single character or have its name spelled out in full. For example: #:: would create a hotkey activated by the hash key, or whatever combination is associated with that symbol (on the US layout, it's Shift+3). ##:: would create a hotkey like the above, but which activates only if you are also holding the Windows key. LWin:: would create a hotkey activated by pressing down the left Windows key without any other modifier keys. The most common modifiers are Ctrl (^), Alt (!), Shift (+) and Win (#). The symbols < and > can be prefixed to any one of the above four modifiers to specify the left or right variant of that key. The modifier combination <^>! corresponds to the AltGr key (if present on your keyboard layout), since the operating system implements it as a combination of LCtrl and RAlt. The other modifiers are: * (wildcard) allows the hotkey to fire even if you are holding modifier keys that the hotkey doesn't include symbols for. ~ (no-suppress) prevents the hotkey from blocking the key's native function. $ (use hook) prevents unintentional loops when sending keys, and in some instances makes the hotkey more reliable. To make a hotkey fire only when you release the key instead of when you press it, use the UP suffix. Related: Hotkey Modifier Symbols, List of Keys Context-sensitive Hotkeys The #HotIf directive can be used to specify a condition that must be met for the hotkey to activate, such as: If a window of a specific app is active when you press the hotkey. If CapsLock is on when you press the hotkey. Any other condition you are able to detect with code. 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." You define the condition by writing an expression which is evaluated whenever you press the hotkey. If the expression evaluates to true, the hotkey's action is executed. The same hotkey can be used multiple times by specifying a different condition for each occurrence of the hotkey, or each hotkey variant. When you press the hotkey, the program executes the first hotkey variant whose condition is met, or the one without a condition (such as the final #c:: in

the example above). If the hotkey's condition isn't met and there is no unconditional variant of the hotkey, the keypress is passed on to the active window as though the hotkey wasn't defined in the first place. For instance, if Notepad isn't active while running the example above, Ctrl+A will perform its normal function (probably Select All). Try making the following changes to the example: Replace WinActive with WinExist so that the hotkeys activate if Notepad is running, even if Notepad doesn't have the focus. Replace the condition with GetKeyState("CapsLock", "T") so that the hotkeys only activate while CapsLock is on. Add another ^a or #c hotkey for some other window, such as your web browser or editor. Note that we use ahk_class so that the example will work on non-English systems, but you can remove it and use the window's title if you wish. Correctly identifying which window you want the hotkey to affect sometimes requires using criteria other than the window title. To learn more, see How to Manage Windows. Related: #HotIf, Expressions, WinActive Custom Combinations A custom combination is a hotkey that combines two keys which aren't normally meant to be used in combination. For example, Numpad0 & Numpad1:: defines a hotkey which activates when you hold Numpad0 and press Numpad1. When you use a key as a prefix in a custom combination, AutoHotkey assumes that you don't want the normal function of the key to activate, since that would interfere with its use as a modifier key. There are two ways to restore the key's normal function: Use another hotkey such as Numpad0::Send "{Numpad0}" to replicate the key's original function. By default, the hotkey will only activate when you release Numpad0, and only if you didn't press Numpad0 and Numpad1 in combination. Prefix the combination with tilde (~), as in ~Numpad0 & Numpad1::. This prevents AutoHotkey from suppressing the normal function of Numpad0, unless you have also defined Numpad0::, in which case the tilde allows the latter hotkey to activate immediately instead of when you release Numpad0. Note: Custom combinations only support combinations of exactly two keys/mouse buttons, and cannot be combined with other modifiers, such as !#^+ for Alt, Win, Ctrl and Shift. Although AutoHotkey does not directly support custom combinations of more than two keys, a similar end result can be achieved by using #HotIf. If you run the example below, pressing Ctrl+CapsLock+Space or Ctrl+Space+CapsLock should show a message: #HotIf GetKeyState ("Ctrl") Space & CapsLock:: CapsLock & Space::MsgBox "Success!" It is necessary to press Ctrl first in this example. This has the advantage that Space and CapsLock perform their normal function if you are not holding Ctrl. Related: Custom Combinations Other Features AutoHotkey's hotkeys have some other features that are worth exploring. While most applications are limited to combinations of Ctrl, Alt, Shift and sometimes Win with one other key (and often not all keys on the keyboard are supported), AutoHotkey isn't so limited. For further reading, see Hotkeys. Any - Methods & Properties | AutoHotkey v2 Any Any is the class at the root of AutoHotkey's type hierarchy. All other types are a sub-type of Any. Any.Prototype defines methods and properties that are applicable to all values and objects (currently excluding ComValue and derived types) unless overridden. The prototype object itself is natively an Object, but has no base and therefore does not identify as an instance of Object. Table of Contents Methods: GetMethod: Retrieves the implementation function of a method. HasBase: Returns true if BaseObj is in Value's chain of base objects. HasMethod: Returns true if the value has a method by this name. HasProp: Returns true if the value has a property by this name. Properties: Base: Retrieves the value's base object. Functions: ObjGetBase: Returns the value's base object. Methods GetMethod Retrieves the implementation function of a method. Value.GetMethod(Name, ParamCount) This method is exactly equivalent to GetMethod(Value, Name, ParamCount), unless overridden. HasBase Returns true if BaseObj is in Value's chain of base objects, otherwise false. Value.HasBase(BaseObj) This method is exactly equivalent to HasBase(Value, BaseObj), unless overridden. HasMethod Returns true if the value has a method by this name, otherwise false. Value.HasMethod(Name, ParamCount) This method is exactly equivalent to HasMethod(Value, Name, ParamCount), unless overridden. HasProp Returns true if the value has a property by this name, otherwise false. Value.HasProp(Name) This method is exactly equivalent to HasProp(Value, Name), unless overridden. Properties Base Retrieves the value's base object. BaseObj := Value.Base For primitive values, the return value is the pre-defined prototype object corresponding to Type(Value). See also: ObjGetBase, ObjSetBase, Obj.Base Functions ObjGetBase Returns the value's base object. BaseObj := ObjGetBase(Value) No meta-functions or property functions are called. Overriding the Base property does not affect the behaviour of this function. If there is no base, the return value is an empty string. Only the Any prototype itself has no base. See also: Base, ObjSetBase, Obj.Base Array Object - Methods & Properties | AutoHotkey v2 Array Object class Array extends Object An Array object contains a list or sequence of values. Values are addressed by their position within the array (known as an array index), where position 1 is the first element. Arrays are often created by enclosing a list of values in brackets. For example: veg := ["Asparagus", "Broccoli", "Cucumber"] Loop veg.Length MsgBox veg[A_Index] A negative index can be used to address elements in reverse, so -1 is the last element, -2 is the second last element, and so on. Attempting to use an array index which is out of bounds (such as zero, or if its absolute value is greater than the Length of the array) is considered an error and will cause an IndexError to be thrown. The best way to add new elements to the array is to call InsertAt or Push. For example: users := Array() users.Push A_UserName MsgBox users[1] An array can also be extended by assigning a larger value to Length. This changes which indices are valid, but Has will show that the new elements have no value. Elements without a value are typically used for variadic calls or by variadic functions, but can be used for any purpose. "ArrayObj" is used below as a placeholder for any Array object, as "Array" is the class itself. In addition to the methods and property inherited from Object, Array objects have the following predefined methods and properties. Table of Contents Static Methods: Call: Creates a new Array containing the specified values. Methods: Clone: Returns a shallow copy of an array. Delete: Removes the value of an array element, leaving the index without a value. Get: Returns the value at a given index, or a default value. Has: Returns true if the specified index is valid and there is a value at that position. InsertAt: Inserts one or more values at a given position. Pop: Removes and returns the last array element. Push: Appends values to the end of an array. RemoveAt: Removes items from an array. __New: Appends items. Equivalent to Push. __Enum: Enumerates array elements. Properties: Length: Retrieves or sets the length of an array. Capacity: Retrieves or sets the current capacity of an array. Default: Defines the default value returned when an element with no value is requested. __Item: Retrieves or sets the value of an array element. Static Methods Call Creates a new Array containing the specified values. ArrayObj := Array(Value, Value2, ..., ValueN) Parameters are defined by __New. Methods Clone Returns a shallow copy of an array. Clone := ArrayObj.Clone() All array elements are copied to the new array. Object references are copied (like with a normal assignment), not the objects themselves. Own properties, own methods and base are copied as per Obj.Clone. Delete Removes the value of an array element, leaving the index without a value. ArrayObj.Delete(Index) Index Type: Integer A valid array index. Returns the removed value (blank if none). Delete does not affect the Length of the array. A ValueError is thrown if Index is out of range. Get Returns the value at a given index, or a default value. Value := ArrayObj.Get (Index , Default) This method does the following: Throw an IndexError if Index is zero or out of range. Return the value at Index, if there is one (see Has). Return the value of the Default parameter, if specified. Return the value of ArrayObj.Default, if defined. Throw an UnsetItemError. When Default is omitted, this is equivalent to ArrayObj[Index], except that __Item is not called. Has Returns true if the specified index is valid and there is a value at that position, otherwise false. ArrayObj.Has(Index) InsertAt Inserts one or more values at a given position. ArrayObj.InsertAt(Index, Value1 , Value2, ... ValueN) Index Type: Integer The position to insert Value1 at. Subsequent values are inserted at Index+1, Index+2, etc. Specifying an index of 0 is the same as specifying Length + 1. Value1 ... One or more values to insert. To insert an array of values, pass theArray* as the last parameter. InsertAt is the counterpart of RemoveAt. Any items previously at or to the right of Index are shifted to the right. Missing parameters are also inserted, but without a value. For example: x := [] x.InsertAt(1, "A", "B") ; => ["A", "B"] x.InsertAt(2, "C") ; => ["A", "C", "B"] ; Missing elements are preserved: x := ["A", , "C"] x.InsertAt(2, "B") ; => ["A", "B", , "C"] x := ["C"] x.InsertAt(1, , "B") ; => [, "B", "C"] A ValueError is thrown if Index is less than -ArrayObj.Length or greater than ArrayObj.Length + 1. For example, with an array of 3 items, Index must be between -3 and 4, inclusive. Pop Removes and returns the last array element. Value := ArrayObj.Pop() All of the following are equivalent: Value := ArrayObj.Pop() Value := ArrayObj.RemoveAt(ArrayObj.Length) Value := ArrayObj.RemoveAt(-1) If the array is empty (Length is 0), an Error is thrown. Push Appends values to the end of an array. ArrayObj.Push(Value, Value2, ..., ValueN) Value ... One or more values to insert. To insert an array of values, pass theArray* as the last parameter. RemoveAt Removes items from an array. ArrayObj.RemoveAt(Index , Length) Index Type: Integer The index of the value or values to remove. Length Type: Integer The length of the range of values to remove. If omitted, one item is removed. If Length is omitted, the removed value is returned (blank if none). Otherwise there is no return value. RemoveAt is the counterpart of InsertAt. A ValueError is thrown if the range indicated by Index and Length is not entirely within the array's current bounds. The remaining items to the right of Pos are shifted to the left by Length (or 1 if omitted). For example: x := ["A", "B"] MsgBox x.RemoveAt(1) ; A MsgBox x[1] ; B x := ["A", , "C"] MsgBox x.RemoveAt(1, 2) ; 1 MsgBox x[1] ; C __New Appends items. Equivalent to Push. ArrayObj.__New(Value, Value2, ..., ValueN) This method exists to support Call, and is not intended to be called directly. See Construction and Destruction. __Enum Enumerates array elements. For Value in ArrayObj For Index, Value in ArrayObj Returns a new enumerator. This method is typically not called directly. Instead, the array object is passed directly to a for-loop, which calls __Enum once and then calls the enumerator once for each iteration of the loop. Each call to the enumerator returns the next array element. The for-loop's variables correspond to the enumerator's parameters, which are: Index Type: Integer The array index, typically the same as A_Index. This is present only in the two-parameter mode. Value The value (if there is no value, Value becomes uninitialized). Properties Length Retrieves or sets the length of an array. Length := ArrayObj.Length ArrayObj.Length := Length The length includes elements which have no value. Increasing the length changes which indices are considered valid, but the new elements have no value (as indicated by Has). Decreasing the length truncates the array. MsgBox ["A", "B", "C"].Length ; 3 MsgBox ["A", , "C"].Length ; 3 Capacity Retrieves or sets the current capacity of an array. ArrayObj.Capacity := MaxItems MaxItems := ArrayObj.Capacity MaxItems Type: Integer The maximum number of elements the array should be able to contain before it must be automatically expanded. If setting a value less than Length, elements are removed. Default Defines the default value returned when an element with no value is requested. ArrayObj.Default := Value This property actually doesn't exist by default, but can be defined by the script. When the script requests an element which has no value, the array checks for this property before throwing an UnsetItemError. It can be implemented by any of the normal means, including a dynamic property or meta-function, but determining which key was queried would require overriding __Item or Get instead. Setting a default value does not prevent an error from being thrown when the index is out of range. __Item Retrieves or sets the value of an array element. Value := ArrayObj[Index] ArrayObj[Index] := Value Index Type: Integer A valid array index; that is, an integer with absolute value between 1 and Length (inclusive). A negative index can be used to address

elements in reverse, so that -1 is the last element, -2 is the second last element, and so on. Attempting to use an index which is out of bounds (such as zero, or if its absolute value is greater than the Length of the array) is considered an error and will cause an IndexError to be thrown. The property name __Item is typically omitted, as shown above, but is used when overriding the property. A_Clipboard - Definition & Usage | AutoHotkey v2 A_Clipboard A_Clipboard is a built-in variable that reflects the current contents of the Windows clipboard if those contents can be expressed as text. Each line of text on A_Clipboard typically ends with carriage return and linefeed (CR+LF), which can be expressed in the script as `r`n. Files (such as those copied from an open Explorer window via Ctrl+C) are considered to be text: They are automatically converted to their filenames (with full path) whenever A_Clipboard is referenced in the script. To extract the files one by one, follow this example: Loop Parse A_Clipboard, "`n", "`r" { Result := MsgBox("File number " A_Index " is " A_LoopField ".`n`nContinue?",, 4) if Result = "No" break } To arrange the filenames in alphabetical order, use the Sort function. To write the filenames on the clipboard to a file, use FileAppend A_Clipboard "`r`n", "C:\My File.txt". To change how long the script will keep trying to open the clipboard -- such as when it is in use by another application -- use #ClipboardTimeout. ClipWait may be used to detect when the clipboard contains data (optionally including non-text data): A_Clipboard := "" ; Start off empty to allow ClipWait to detect when the text has arrived. Send "^c" ClipWait ; Wait for the clipboard to contain text. MsgBox "Control-C copied the following contents to the clipboard:`n`n" A_Clipboard Related ClipboardAll: For operating upon everything on the clipboard (such as pictures and formatting). OnClipboardChange: For detecting and responding to clipboard changes. Examples Gives the clipboard entirely new contents. A_Clipboard := "my text" Empties the clipboard. A_Clipboard := "" Converts any copied files, HTML, or other formatted text to plain text. A_Clipboard := A_Clipboard Appends some text to the clipboard. A_Clipboard .= " Text to append." Replaces all occurrences of ABC with DEF (also converts the clipboard to plain text). A_Clipboard := StrReplace(A_Clipboard, "ABC", "DEF") Clipboard utilities written in AutoHotkey v1: Deluxe Clipboard: Provides unlimited number of private, named clipboards to Copy, Cut, Paste, Append or CutAppend of selected text. ClipStep: Control multiple clipboards using only the keyboard's Ctrl-X-C-V. A_HotkeyModifierTimeout - Syntax & Usage | AutoHotkey v2 A_HotkeyModifierTimeout Affects the behavior of Send with hotkey modifiers: Ctrl, Alt, Win, and Shift. The built-in variable A_HotkeyModifierTimeout defines how long after a hotkey is pressed that its modifier keys are assumed to still be held down. This is used by Send to determine whether to push the modifier keys back down after having temporarily released them. Value Type: Integer The length of the interval in milliseconds. The value can be -1 so that it never times out (modifier keys are always pushed back down after the Send), or 0 so that it always times out (modifier keys are never pushed back down). The default value is 50. Remarks This variable has no effect when: Hotkeys send their keystrokes via the SendInput or SendPlay methods. This is because those methods postpone the user's physical pressing and releasing of keys until after the Send completes. The script has the keyboard hook installed (you can see if your script uses the hook via the "View->Key history" menu item in the main window, or via the KeyHistory function). This is because the hook can keep track of which modifier keys (Alt/Ctrl/Win/Shift) the user is physically holding down and doesn't need to use the timeout. To illustrate the effect of this variable, consider this example: ^!a::Send "abc". When the Send function executes, the first thing it does is release Ctrl and Alt so that the characters get sent properly. After sending all the keys, the function doesn't know whether it can safely push back down Ctrl and Alt (to match whether the user is still holding them down). But if less than the specified number of milliseconds have elapsed, it will assume that the user hasn't had a chance to release the keys yet and it will thus push them back down to match their physical state. Otherwise, the modifier keys will not be pushed back down and the user will have to release and press them again to get them to modify the same or another key. The timeout should be set to a value less than the amount of time that the user typically holds down a hotkey's modifiers before releasing them. Otherwise, the modifiers may be restored to the down position (get stuck down) even when the user isn't physically holding them down. You can reduce or eliminate the need for this variable with one of the following: Install the keyboard hook by calling InstallKeybdHook. Use the SendInput or SendPlay methods rather than the traditional SendEvent method. When using the traditional SendEvent method, reduce SetKeyDelay to 0 or -1, which should help because it sends the keystrokes more quickly. Related GetKeyState Examples Sets the hotkey modifier timeout to 100 ms instead of 50 ms. A_HotkeyModifierTimeout := 100 A_MaxHotkeysPerInterval - Syntax & Usage | AutoHotkey v2 A_MaxHotkeysPerInterval The A_MaxHotkeysPerInterval and A_HotkeyInterval variables control the rate of hotkey activations beyond which a warning dialog will be displayed. VariableMeaningDefault ValueType A_MaxHotkeysPerInterval The maximum number of hotkeys that can be pressed within the interval without triggering a warning dialog. 70 Integer A_HotkeyInterval The length of the interval in milliseconds. 2000 Integer Remarks These built-in variables should usually be assigned values when the script starts (if the default values are not suitable), but the script can get or set their values at any time. Care should be taken not to make the above too lenient because if you ever inadvertently introduce an infinite loop of keystrokes (via a Send function that accidentally triggers other hotkeys), your computer could become unresponsive due to the rapid flood of keyboard events. As an oversimplified example, the hotkey ^c::Send "^c" would produce an infinite loop of keystrokes. To avoid this, add the $ prefix to the hotkey definition (e.g. $^c::) so that the hotkey cannot be triggered by the Send function. The limit might be reached by means other than an infinite loop, such as: Key-repeat when the limit is too low relative to the key-repeat rate, or the system is under heavy load. Keyboard or mouse hardware which sends input events more rapidly than the typical key-repeat rate. For example, tilting the wheel left or right on some mice sends a rapid flood of events which may reach the limit for hotkeys such as WheelLeft:: and WheelRight::. To disable the warning dialog entirely, assign A_HotkeyInterval := 0. Examples Allows a maximum of 200 hotkeys to be pressed within 2000 ms without triggering a warning dialog. A_HotkeyInterval := 2000 ; This is the default value (milliseconds). A_MaxHotkeysPerInterval := 200 A_MenuMaskKey - Syntax & Usage | AutoHotkey v2 A_MenuMaskKey Controls which key is used to mask Win or Alt keyup events. Value Type: String A vkNNscNNN sequence identifying the virtual key code (NN) and scan code (NNN), in hexadecimal, or an empty string if masking is disabled. The script can also assign a key name, vkNN sequence or scNNN sequence, in which case generally either the VK or SC code is left as zero until the key is sent, then determined automatically. Assigning "vk00sc000" disables masking and is equivalent to assigning "". The returned string is always a vkNNscNNN sequence if enabled or "" if disabled, regardless of how it was assigned. All vkNNscNNN sequences and all non-zero vkNN or scNNN sequences are permitted, but some combinations may fail to suppress the menu. Any other invalid keys cause a ValueError to be thrown. Remarks The mask key is sent automatically to prevent the Start menu or the active window's menu bar from activating at unexpected times. The default mask key is Ctrl. This variable can be used to change the mask key to a key with fewer side effects. Good candidates are virtual key codes which generally have no effect, such as vkE8, which Microsoft documents as "unassigned", or vkFF, which is reserved to mean "no mapping" (a key which has no function). Some values, such as zero VK with non-zero SC, may fail to suppress the Start menu. Key codes are not required to match an existing key. Note: Microsoft can assign an effect to an unassigned key code at any time. For example, vk07 was once undefined and safe to use, but since Windows 10 1909 it is reserved for opening the game bar. This setting is global, meaning that it needs to be specified only once to affect the behavior of the entire script. Hotkeys: If a hotkey is implemented using the keyboard hook or mouse hook, the final keypress may be invisible to the active window and the system. If the system was to detect only a Win or Alt keydown and keyup with no intervening keypress, it would usually activate a menu. To prevent this, the keyboard or mouse hook may automatically send the mask key. Pressing a hook hotkey causes the next Alt or Win keyup to be masked if all of the following conditions are met: The hotkey is suppressed (it lacks the tilde modifier). Alt or Win is logically down when the hotkey is pressed. The modifier is physically down or the hotkey requires the modifier to activate. For example, $#a:: in combination with AppsKey::RWin causes masking when Menu+A is pressed, but Menu on its own is able to open the Start Menu. Alt is not masked if Ctrl was down when the hotkey was pressed, since Ctrl+Alt does not activate the menu bar. Win is not masked if the most recent Win keydown was modified with Ctrl, Shift or Alt, since the Start Menu does not normally activate in those cases. However, key-repeat occurs even for Win if it was the last key physically pressed, so it can be hard to predict when the most recent Win keydown was. Either the keyboard hook is not installed (i.e. for a mouse hotkey), or there have been no other (unsuppressed) keydown or keyup events since the last Alt or Win keydown. Note that key-repeat occurs even for modifier keys and even after sending other keys, but only for the last physically pressed key. Mouse hotkeys may send the mask key immediately if the keyboard hook is not installed. Hotkeys with the tilde modifier are not intended to block the native function of the key, so they do not cause masking. Hotkeys like ~#a:: still suppress the menu, since the system detects that Win has been used in combination with another key. However, mouse hotkeys and both Win themselves (~LWin:: and ~RWin::) do not suppress the Start Menu. The Start Menu (or the active window's menu bar) can be suppressed by sending any keystroke. The following example disables the ability for the left Win to activate the Start Menu, while still allowing its use as a modifier: ~LWin::Send "{Blind}{vkE8}" Send: Send, ControlSend and related often release modifier keys as part of their normal operation. For example, the hotkey <#a::SendText Address usually must release the left Win prior to sending the contents of Address, and press the left Win back down afterward (so that other Win key combinations continue working). The mask key may be sent in such cases to prevent a Win or Alt keyup from activating a menu. Related See this thread for background information. Examples Basic usage. A_MenuMaskKey := "vkE8" ; Change the masking key to something unassigned such as vkE8. #Space::Run A_ScriptDir ; An additional Ctrl keystroke is not triggered. Shows in detail how this variable causes vkFF to be sent instead of LControl. A_MenuMaskKey := "vkFF" ; vkFF is no mapping. #UseHook #Space:: !Space:: { KeyWait "LWin" KeyWait "RWin" KeyWait "Alt" KeyHistory } {...} (block) - Syntax & Usage | AutoHotkey v2 {...} (block) Blocks are one or more statements enclosed in braces. Typically used with function definitions and control flow statements. { Statements } Remarks A block is used to bind two or more statements together. It can also be used to change which If statement an Else statement belongs to, as in this example where the block forces the Else statement to belong to the first If statement rather than the second: if (Var1 = 1) { if (Var2 = "abc") Sleep 1 } else return Although blocks can be used anywhere, currently they are only meaningful when used with function definitions, If, Else, Loop statements, Try, Catch or Finally. If any of the control flow statements mentioned above has only a single statement, that statement need not be enclosed in a block (this does not work for function definitions). However, there may be cases where doing so enhances the readability or maintainability of the script. A block may be empty (contain zero statements), which may be useful in cases where you want to comment out the contents of the block without removing the block itself. One True Brace (OTB, K&R style): The OTB style may optionally be used in the following places: function definitions, If, Else, Loop, While, For, Try, Catch, and Finally. This style puts the block's opening brace on the same line as

the block's controlling statement rather than underneath on a line by itself. For example: MyFunction(x, y) { ... } if (x < y) { ... } else { ... } Loop RepeatCount { ... } While x < y { ... } For k, v in obj { ... } Try { ... } Catch Error { ... } Finally { .... } Similarly, a statement may exist to the right of a brace (except the open-brace of the One True Brace style). For example: if (x = 1) { MsgBox "This line appears to the right of an opening brace. It executes whenever the IF-statement is true." MsgBox "This is the next line." } MsgBox "This line appears to the right of a closing brace. It executes unconditionally." Related Function Definitions, Control Flow Statements, If, Else, Loop Statements, Try, Catch, Finally Examples By enclosing the two statements MsgBox "test1" and Sleep 5 with braces, the If statement knows that it should execute both if x is equal to 1. if (x = 1) { MsgBox "test1" Sleep 5 } else MsgBox "test2" BlockInput - Syntax & Usage | AutoHotkey v2 BlockInput Disables or enables the user's ability to interact with the computer via keyboard and mouse. BlockInput OnOff BlockInput SendMouse BlockInput MouseMove Parameters OnOff Type: String or Integer This mode blocks all user inputs unconditionally. Specify one of the following values: On or 1 (true): The user is prevented from interacting with the computer (mouse and keyboard input has no effect). Off or 0 (false): Input is re-enabled. SendMouse Type: String This mode only blocks user inputs while specific send and/or mouse functions are in progress. Specify one of the following words: Send: The user's keyboard and mouse input is ignored while a SendEvent is in progress (including Send and SendText if SendMode "Event" has been used). This prevents the user's keystrokes from disrupting the flow of simulated keystrokes. When the Send finishes, input is re-enabled (unless still blocked by a previous use of BlockInput "On"). Mouse: The user's keyboard and mouse input is ignored while a Click, MouseMove, MouseClick, or MouseClickDrag is in progress (the traditional SendEvent mode only). This prevents the user's mouse movements and clicks from disrupting the simulated mouse events. When the mouse action finishes, input is re-enabled (unless still blocked by a previous use of BlockInput "On"). SendAndMouse: A combination of the above two modes. Default: Turns off both the Send and the Mouse modes, but does not change the current state of input blocking. For example, if BlockInput "On" is currently in effect, using BlockInput "Default" will not turn it off. MouseMove Type: String This mode only blocks the mouse cursor movement. Specify one of the following words: MouseMove: The mouse cursor will not move in response to the user's physical movement of the mouse (DirectInput applications are a possible exception). When a script first uses this function, the mouse hook is installed (if it is not already). The mouse hook will stay installed until the next use of the Suspend or Hotkey function, at which time it is removed if not required by any hotkeys or hotstrings (see #Hotstring NoMouse). MouseMoveOff: Allows the user to move the mouse cursor. Remarks All three BlockInput modes (OnOff, SendMouse and MouseMove) operate independently of each other. For example, BlockInput "On" will continue to block input until BlockInput "Off" is used, even if one of the words from SendMouse is also in effect. Another example is, if BlockInput "On" and BlockInput "MouseMove" are both in effect, mouse movement will be blocked until both are turned off. Note: The OnOff and SendMouse modes might have no effect if UAC is enabled or the script has not been run as administrator. For more information, refer to the FAQ. In preference to BlockInput, it is often better to use SendMode "Input" or SendMode "Play" so that keystrokes and mouse clicks become uninterruptible. This is because unlike BlockInput, those modes do not discard what the user types during the send; instead, those keystrokes are buffered and sent afterward. Avoiding BlockInput also avoids the need to work around sticking keys as described in the next paragraph. If BlockInput becomes active while the user is holding down keys, it might cause those keys to become "stuck down". This can be avoided by waiting for the keys to be released prior to turning BlockInput on, as in this example: ^!p:: { KeyWait "Control" ; Wait for the key to be released. Use one KeyWait for each of the hotkey's modifiers. KeyWait "Alt" BlockInput True ; ... send keystrokes and mouse clicks ... BlockInput False } When BlockInput is in effect, user input is blocked but AutoHotkey can simulate keystrokes and mouse clicks. However, pressing Ctrl+Alt+Del will re-enable input due to a Windows API feature. Certain types of hook hotkeys can still be triggered when BlockInput is on. Examples include MButton (mouse hook) and LWin & Space (keyboard hook with explicit prefix rather than modifiers "$#"). Input is automatically re-enabled when the script closes. Related SendMode, Send, Click, MouseMove, MouseClick, MouseClickDrag Examples Opens Notepad and pastes time/date by sending F5 while BlockInput is turned on. Note that BlockInput may only work if the script has been run as administrator. BlockInput true Run "notepad" WinWaitActive "ahk_class Notepad" Send "{F5}" ; pastes time and date BlockInput false Break - Syntax & Usage | AutoHotkey v2 Break Exits (terminates) any type of loop statement. Break LoopLabel Parameters LoopLabel LoopLabel identifies which loop this statement should apply to; either by label name or numeric nesting level. If omitted or 1, this statement applies to the innermost loop in which it is enclosed. If a label is specified, it must point directly at a loop statement. LoopLabel must be a constant value - variables and expressions are not supported, with the exception of a single literal number or quoted string enclosed in parentheses. For example: break("outer") Remarks The use of Break and Continue are encouraged over Goto since they usually make scripts more readable and maintainable. Related Continue, Loop, While-loop, For-loop, Blocks, Labels Examples Breaks the loop if var is greater than 25. Loop { ; ... if (var > 25) break ; ... if (var <= 5) continue } Breaks the outer loop from within a nested loop. outer: Loop 3 { x := A_Index Loop 3 { if (x*A_Index = 6) break outer ; Equivalent to break 2 or goto break_outer. MsgBox x "," A_Index } } break_outer: ; For goto. Buffer Object - Definition & Usage | AutoHotkey v2 Buffer Object class Buffer extends Object Encapsulates a block of memory for use with advanced techniques such as DllCall, structures, StrPut and raw file I/O. Buffer objects are typically created by calling Buffer(), but can also be returned by FileRead with the "RAW" option. BufferObj := Buffer(ByteCount) ClipboardAll returns a sub-type of Buffer, also named ClipboardAll. class ClipboardAll extends Buffer "BufferObj" is used below as a placeholder for any Buffer object, as "Buffer" is the class itself. In addition to the methods and property inherited from Object, Buffer objects have the following predefined properties. Table of Contents Buffer-like Objects Static Methods: Call: Creates a Buffer. Methods: __New: Allocates or reallocates the buffer and optionally fills it. Properties: Ptr: Retrieves the buffer's current memory address. Size: Retrieves or sets the buffer's size, in bytes. Related Examples Buffer-like Objects Some built-in functions accept a Buffer object in place of an address - see the Related section for links. These functions also accept any other object which has Ptr and Size properties, but are optimized for the native Buffer object. In most cases, passing a Buffer object is safer than passing an address, as the function can read the buffer size to ensure that it does not attempt to access any memory location outside of the buffer. One exception is that DllCall calls functions outside of the program; in those cases, it may be necessary to explicitly pass the buffer size to the function. Static Methods Call Creates a Buffer. BufferObj := Buffer(ByteCount, FillByte) ByteCount Type: Integer The number of bytes to allocate. Corresponds to Buffer.Size. If omitted, the Buffer is created with a null (zero) Ptr and zero Size. FillByte Type: Integer Specify a number between 0 and 255 to set each byte in the buffer to that number. This should generally be omitted in cases where the buffer will be written into without first being read, as it has a time-cost proportionate to the number of bytes. If omitted, the memory of the buffer is not initialized; the value of each byte is arbitrary. A MemoryError is thrown if the memory could not be allocated, such as if ByteCount is unexpectedly large or the system is low on virtual memory. Parameters are defined by __New. Methods __New Allocates or reallocates the buffer and optionally fills it. BufferObj.__New(ByteCount, FillByte) This method exists to support Call, and is not usually called directly. See Construction and Destruction. Specify ByteCount to allocate, reallocate or free the buffer. This is equivalent to assigning Size. Specify FillByte to fill the buffer with the given numeric byte value, overwriting any existing content. If both parameters are omitted, this method has no effect. Properties Ptr Retrieves the buffer's current memory address. Ptr := BufferObj.Ptr Type: Integer Any address returned by this property becomes invalid when the buffer is freed or reallocated. Invalid addresses must not be used. The buffer is not freed until the Buffer object's reference count reaches zero, but it is reallocated when its Size is changed. Size Retrieves or sets the buffer's size, in bytes. ByteCount := BufferObj.Size BufferObj.Size := ByteCount Type: Integer The buffer's address typically changes whenever its size is changed. If the size is decreased, the data within the buffer is truncated, but the remaining bytes are preserved. If the size is increased, all data is preserved and the values of any new bytes are arbitrary (they are not initialized, for performance reasons). A MemoryError is thrown if the memory could not be allocated, such as if ByteCount is unexpectedly large or the system is low on virtual memory. This property always returns the exact value it was given either by __New or by a previous assignment. Related DllCall, NumPut, NumGet, StrPut, StrGet, File.RawRead, File.RawWrite, ClipboardAll Examples Use a Buffer to receive a string from an external function via DllCall. max_chars := 11 ; Allocate a buffer for use with the Unicode version of wsprintf. bufW := Buffer(max_chars*2) ; Print a UTF-16 string into the buffer with wsprintfW(). DllCall("wsprintfW", "Ptr", bufW, "Str", "0x%08x", "UInt", 4919, "CDecl") ; Retrieve the string from bufW and show it. MsgBox StrGet(bufW, "UTF-16") ; Or just StrGet(bufW). ; Allocate a buffer for use with the ANSI version of wsprintf. bufA := Buffer(max_chars) ; Print an ANSI string into the buffer with wsprintfA(). DllCall("wsprintfA", "Ptr", bufA, "AStr", "0x%08x", "UInt", 4919, "CDecl") ; Retrieve the string from bufA (converted to the native format), and show it. MsgBox StrGet(bufA, "CP0") CallbackCreate - Syntax & Usage | AutoHotkey v2 CallbackCreate Creates a machine-code address that when called, redirects the call to a function in the script. Address := CallbackCreate(Function , Options, ParamCount) Parameters Function Type: Function Object A function object to call automatically whenever Address is called. The function also receives the parameters that were passed to Address. A closure or bound function can be used to differentiate between multiple callbacks which all call the same script function. The callback retains a reference to the function object, and releases it when the script calls CallbackFree. Options Type: String Specify zero or more of the following words or characters. Separate each option from the next with a space (e.g. "C Fast"). Fast or F: Avoids starting a new thread each time Function is called. Although this performs better, it must be avoided whenever the thread from which Address is called varies (e.g. when the callback is triggered by an incoming message). This is because Function will be able to change global settings such as A_LastError and the last-found window for whichever thread happens to be running at the time it is called. For more information, see Remarks. CDecl or C: Makes Address conform to the "C" calling convention. This is typically omitted because the standard calling convention is much more common for callbacks. This option is ignored by 64-bit versions of AutoHotkey, which use the x64 calling convention. &: Causes the address of the parameter list (a single integer) to be passed to Function instead of the individual parameters. Parameter values can be retrieved by using NumGet. When using the standard 32-bit calling convention, ParamCount must specify the size of the parameter list in DWORDs (the number of bytes divided by 4). ParamCount Type: Integer The number of parameters that Address's caller will pass to it. If omitted, it defaults to Function.MinParams, which is usually the number of mandatory parameters in the definition of Function. In either case, ensure that the caller passes exactly this number of parameters. Return Value Type: Integer CallbackCreate returns a machine-code address. This address is typically passed to an external function via DllCall or placed in a struct using NumPut, but can also be called directly by DllCall. Passing the address to CallbackFree will delete the

callback. Error Handling This function fails and throws an exception under any of the following conditions: Function is not an object, or has neither a MinParams property nor a Call method. Function has a MinParams property which exceeds the number of parameters that the callback will supply. ParamCount is negative. ParamCount is omitted and: 1) Function has no MinParams property; or 2) the & option is used with the standard 32-bit calling convention. The Callback Function's Parameters A function assigned to a callback address may accept up to 31 parameters. Optional parameters are permitted, which is useful when the function is called by more than one caller. Interpreting the parameters correctly requires some understanding of how the x86 calling conventions work. Since AutoHotkey does not have typed parameters, the callback's parameter list is assumed to consist of integers, and some reinterpretation may be required. AutoHotkey 32-bit: All incoming parameters are unsigned 32-bit integers. Smaller types are padded out to 32 bits, while larger types are split into a series of 32-bit parameters. If an incoming parameter is intended to be a signed integer, any negative numbers can be revealed by following either of the following examples: ; Method #1 if (wParam > 0x7FFFFFFF) wParam := -(~wParam) - 1 ; Method #2: Relies on the fact that AutoHotkey natively uses signed 64-bit integers. wParam := wParam << 32 >> 32 AutoHotkey 64-bit: All incoming parameters are signed 64-bit integers. AutoHotkey does not natively support unsigned 64-bit integers. Smaller types are padded out to 64 bits, while larger types are always passed by address. AutoHotkey 32-bit/64-bit: If an incoming parameter is intended to be 8-bit or 16-bit (or 32-bit on x64), the upper bits of the value might contain "garbage" which can be filtered out by using bitwise-and, as in the following examples: Callback(UCharParam, UShortParam, UIntParam) { UCharParam &= 0xFF UShortParam &= 0xFFFF UIntParam &= 0xFFFFFFFF ;... } If an incoming parameter is intended by its caller to be a string, what it actually receives is the address of the string. To retrieve the string itself, use StrGet: MyString := StrGet (MyParameter) If an incoming parameter is the address of a structure, the individual members may be extracted by following the steps at DllCall structures. Receiving parameters by address: If the & option is used, the function receives the address of the first callback parameter. For example: callback := CallbackCreate (TheFunc, "F&", 3) ; Parameter list size must be specified for 32-bit. DllCall(callback, "float", 10.5, "int64", 42) TheFunc(params) { MsgBox NumGet(params, 0, "float") ", " NumGet(params, A_PtrSize, "int64") } Most callbacks in 32-bit programs use the stdcall calling convention, which requires a fixed number of parameters. In those cases, ParamCount must be set to the size of the parameter list, where Int64 and Double count as two 32-bit parameters. With Cdecl or the 64-bit calling convention, ParamCount has no effect. What the Function Should Return If the function uses Return without any parameters, or it specifies a blank value such as "" (or it never uses Return at all), 0 is returned to the caller of the callback. Otherwise, the function should return an integer, which is then returned to the caller. AutoHotkey 32-bit truncates return values to 32-bit, while AutoHotkey 64-bit supports 64-bit return values. Returning structs larger than this (by value) is not supported. Fast vs. Slow The default/slow mode causes the function to start off fresh with the default values for settings such as SendMode and DetectHiddenWindows. These defaults can be changed during script startup. By contrast, the fast mode inherits global settings from whichever thread happens to be running at the time the function is called. Furthermore, any changes the function makes to global settings (including the last-found window) will go into effect for the current thread. Consequently, the fast mode should be used only when it is known exactly which thread(s) the function will be called from. To avoid being interrupted by itself (or any other thread), a callback may use Critical as its first line. However, this is not completely effective when the function is called indirectly via the arrival of a message less than 0x0312 (increasing Critical's interval may help). Furthermore, Critical does not prevent the function from doing something that might indirectly result in a call to itself, such as calling SendMessage or DllCall. CallbackFree Deletes a callback and releases its reference to the function object. CallbackFree(Address) Each use of CallbackCreate allocates a small amount of memory (32 or 48 bytes plus system overhead). Since the OS frees this memory automatically when the script exits, any script that allocates a small, fixed number of callbacks can get away with not explicitly freeing the memory. However, if the function object held by the callback is of a dynamic nature (such as a closure or bound function), it can be especially important to free the callback when it is no longer needed; otherwise, the function object will not be released. Related DllCall, OnMessage, OnExit, OnClipboardChange, Sort's callback, Critical, PostMessage, SendMessage, Functions, List of Windows Messages, Threads Examples Displays a summary of all top-level windows. EnumAddress := CallbackCreate(EnumWindowsProc, "Fast") ; Fast-mode is okay because it will be called only from this thread. DetectHiddenWindows True ; Due to fast-mode, this setting will go into effect for the callback too. ; Pass control to EnumWindows(), which calls the callback repeatedly: DllCall("EnumWindows", "Ptr", EnumAddress, "Ptr", 0) MsgBox Output ; Display the information accumulated by the callback. EnumWindowsProc(hwnd, lParam) { global Output win_title := WinGetTitle(hwnd) win_class := WinGetClass(hwnd) if win_title Output .= "HWND: " hwnd "`tTitle: " win_title "`tClass: " win_class "`n" return true ; Tell EnumWindows() to continue until all windows have been enumerated. } Demonstrates how to subclass a GUI window by redirecting its WindowProc to a new WindowProc in the script. In this case, the background color of a text control is changed to a custom color. TextBackgroundColor := 0xFFBBBB ; A custom color in BGR format. TextBackgroundBrush := DllCall("CreateSolidBrush", "UInt", TextBackgroundColor) MyGui := Gui() Text := MyGui.Add("Text",, "Here is some text that is given`na custom background color.") ; 64-bit scripts must call SetWindowLongPtr instead of SetWindowLong: SetWindowLong := A_PtrSize=8 ? "SetWindowLongPtr" : "SetWindowLong" WindowProcNew := CallbackCreate(WindowProc) ; Avoid fast-mode for subclassing. WindowProcOld := DllCall (SetWindowLong, "Ptr", MyGui.Hwnd, "Int", -4 ; -4 is GWL_WNDPROC , "Ptr", WindowProcNew, "Ptr") ; Return value must be set to "Ptr" or "UPtr" vs. "Int". MyGui.Show WindowProc(hwnd, uMsg, wParam, lParam) { Critical if (uMsg = 0x0138 && lParam = Text.Hwnd) ; 0x0138 is WM_CTLCOLORSTATIC. { DllCall("SetBkColor", "Ptr", wParam, "UInt", TextBackgroundColor) return TextBackgroundBrush ; Return the HBRUSH to notify the OS that we altered the HDC. } ; Otherwise (since above didn't return), pass all unhandled events to the original WindowProc. return DllCall("CallWindowProc", "Ptr", WindowProcOld, "Ptr", hwnd, "UInt", uMsg, "Ptr", wParam, "Ptr", lParam) } CaretGetPos - Syntax & Usage | AutoHotkey v2 CaretGetPos Retrieves the current position of the caret (text insertion point). CaretFound := CaretGetPos(&OutputVarX, &OutputVarY) Parameters &OutputVarX, &OutputVarY Type: VarRef References to the output variables in which to store the X and Y coordinates. The retrieved coordinates are relative to the active window's client area unless overridden by using CoordMode or A_CoordModeCaret. Return Value Type: Integer (boolean) If there is no active window or the caret position cannot be determined, the function returns 0 (false) and the output variables are made blank. The function returns 1 (true) if the system returned a caret position, but this does not necessarily mean a caret is visible. Remarks Any of the output variables may be omitted if the corresponding information is not needed. Note that some windows (e.g. certain versions of MS Word) report the same caret position regardless of its actual position. Related CoordMode, A_CoordModeCaret Examples Allows the user to move the caret around to see its current position displayed in an auto-update tooltip. SetTimer WatchCaret, 100 WatchCaret() { if CaretGetPos(&x, &y) ToolTip "X" x " Y" y, x, y - 20 else ToolTip "No caret" } Catch - Syntax & Usage | AutoHotkey v2 Catch Specifies the code to execute if a value or error is thrown during execution of a try statement. Catch ErrorClass as OutputVar { Statements } Parameters ErrorClass Type: Class The class of value that should be caught, such as Error, TimeoutError or MyCustomError. This can also be a comma-delimited list of classes. Classes must be specified by their exact full name and not an arbitrary expression, as the Prototype of each class is resolved at load time. Any built-in or user-defined class can be used, even if it does not derive from Error. If no classes are specified, the default is Error. To catch anything at all, use Catch Any. A load-time error is displayed if an invalid class name is used, or if a class is inaccessible due to the presence of a local variable with the same name. OutputVar Type: Variable The output variable in which to store the thrown value, which is typically an Error object. This cannot be a dynamic variable. If omitted, the thrown value cannot be accessed directly, but can still be re-thrown by using Throw with no parameter. Statements The statements to execute if a value or error is thrown. Braces are generally not required if only a single statement is used. For details, see {...} (block). Remarks Multiple catch statements can be used one after the other, with each one specifying a different class (or multiple classes). If the value is not an instance of any of the listed classes, it is not caught by this try-catch, but might be caught by one further up the call stack. Every use of catch must belong to (be associated with) a try statement above it. A catch always belongs to the nearest unclaimed try statement above it unless a block is used to change that behavior. The parameter list may optionally be enclosed in parentheses, in which case the space or tab after Catch is optional. Catch may optionally be followed by else, which is executed if no exception was thrown within the associated try block. The One True Brace (OTB) style may optionally be used. For example: try { ... } catch Error { ... } Load-time errors cannot be caught, since they occur before the try statement is executed. Related Try, Throw, Error Object, Else, Finally, Blocks, OnError Examples See Try. Chr - Syntax & Usage | AutoHotkey v2 Chr Returns the string (usually a single character) corresponding to the character code indicated by the specified number. String := Chr(Number) Parameters Number Type: Integer A Unicode character code between 0 and 0x10FFFF. Return Value Type: String The string corresponding to Number. This is always a single Unicode character, but for practical reasons, Unicode supplementary characters (where Number is in the range 0x10000 to 0x10FFFF) are counted as two characters. That is, the length of the return value as reported by StrLen may be 1 or 2. For further explanation, see String Encoding. If Number is 0, the return value is a string containing a binary null character, not an empty (zero-length) string. This can be safely assigned to a variable, passed to a function or concatenated with another string. However, some built-in functions may "see" only the part of the string preceding the first null character. Remarks The range and meaning of character codes depends on which string encoding is in use. Currently all AutoHotkey v2 executables are built for Unicode, so this function always accepts a Unicode character code and returns a Unicode (UTF-16) string. Common character codes include 9 (tab), 10 (linefeed), 13 (carriage return), 32 (space), 48-57 (the digits 0-9), 65-90 (uppercase A-Z), and 97-122 (lowercase a-z). Related Ord Examples Reports the string corresponding to the character code 116. MsgBox Chr(116) ; Reports "t". Class Object - Methods & Properties | AutoHotkey v2 Class Object class Class extends Object A Class object represents a class definition; it contains static methods and properties. Each class object is based on whatever class it extends, or Object if not specified. The global class object Object is based on Class.Prototype, which is based on Object.Prototype, so classes can inherit methods and properties from any of these base objects. "Static" methods and properties are any methods and properties which are owned by the class object itself (and therefore do not apply to a specific instance), while methods and properties for instances of the class are owned by the class's Prototype. "ClassObj" is used below as a placeholder for any class object, as "Class" is the Class class itself. Ordinarily, one refers to a class object by the name given in its class definition. Table of Contents Methods: Call: Constructs a new instance of the class. Properties: Prototype: Retrieves or sets the object on which all instances of the class are based. Methods Call Constructs a new instance of

the class. Obj := ClassObj(Params*) Obj := ClassObj.Call(Params*) This static method is typically inherited from the Object, Array or Map class. It performs the following functions: Allocate memory and initialize the binary structure of the object, which depends on the object's native type (e.g. whether it is an Array or Map, or just an Object). Set the base of the new object to ClassObj.Prototype. Call the new object's __Init method, if it has one. This method is automatically created by class definitions; it contains all instance variable initializers defined within the class body. Call the new object's __New method, if it has one. All parameters passed to Call are forwarded on to __New. Return the new object. Call can be overridden within a class definition by defining a static method, such as static Call(). This allows classes to modify or prevent the construction of new instances. Note that Class() (literally using "Class" in this case) can be used to construct a new Class object based on Class.Prototype. However, this new object initially has no Call method as it is not a subclass of Object. It can become a subclass of Object by assigning to Base, or the Call method can be reimplemented or copied from another class. A Prototype must also be created and assigned to the class before it can be instantiated with the standard Call method. Properties Prototype Retrieves or sets the object on which all instances of the class are based. Proto := ClassObj.Prototype ClassObj.Prototype := Proto By default, the class's Prototype contains all instance methods and dynamic properties defined within the class definition, and can be used to retrieve references to methods or property getters/setters or define new ones. The script can also define new value properties, which act as default property values for all instances. A class's Prototype is normally based on the Prototype of its base class, so ClassObj.Prototype.base == ClassObj.base.Prototype. Prototype is automatically defined as an own property of any class object created by a class definition. Click - Syntax & Usage | AutoHotkey v2 Click Clicks a mouse button at the specified coordinates. It can also hold down a mouse button, turn the mouse wheel, or move the mouse. Click Options Parameters Options Specify zero or more of the following components: Coords, WhichButton, ClickCount, DownOrUp, and/or Relative. Separate each component from the next with at least one space, tab, and/or comma (within a string or as separate parameters). The components can appear in any order except ClickCount, which must occur somewhere to the right of Coords, if present. Coords: The X and Y coordinates to which the mouse cursor is moved prior to clicking. For example, Click "100 200" clicks the left mouse button at a specific position. Coordinates are relative to the active window's client area unless CoordMode was used to change that. If omitted, the cursor's current position is used. WhichButton: Left (default), Right, Middle (or just the first letter of each of these); or the fourth or fifth mouse button (X1 or X2). For example, Click "Right" clicks the right mouse button at the mouse cursor's current position. Left and Right correspond to the primary button and secondary button. If the user swaps the buttons via system settings, the physical positions of the buttons are swapped but the effect stays the same. WhichButton can also be WheelUp or WU to turn the wheel upward (away from you), or WheelDown or WD to turn the wheel downward (toward you). WheelLeft (or WL) or WheelRight (or WR) may also be specified. For ClickCount, specify the number of notches to turn the wheel. However, some applications do not obey a ClickCount value higher than 1 for the mouse wheel. For them, use the Click function multiple times by means such as Loop. ClickCount: The number of times to click the mouse. For example, Click 2 performs a double-click at the mouse cursor's current position. If omitted, the button is clicked once. If Coords is specified, ClickCount must appear after it. Specify zero (0) to move the mouse without clicking; for example, Click "100 200 0". DownOrUp: This component is normally omitted, in which case each click consists of a down-event followed by an up-event. Otherwise, specify the word Down (or the letter D) to press the mouse button down without releasing it. Later, use the word Up (or the letter U) to release the mouse button. For example, Click "Down" presses down the left mouse button and holds it. Relative: The word Rel or Relative treats the specified X and Y coordinates as offsets from the current mouse position. In other words, the cursor will be moved from its current position by X pixels to the right (left if negative) and Y pixels down (up if negative). Remarks The Click function uses the sending method set by SendMode. To override this mode for a particular click, use a specific Send function in combination with {Click}, as in this example: SendEvent "{Click 100 200}". To perform a shift-click or control-click, clicking via Send is generally easiest. For example: Send "+{Click 100 200}" ; Shift+LeftClick Send "^{Click 100 200 Right}" ; Control+RightClick Unlike Send, the Click function does not automatically release the modifier keys (Ctrl, Alt, Shift, and Win). For example, if Ctrl is currently down, Click would produce a control-click but Send "{Click}" would produce a normal click. The SendPlay mode is able to successfully generate mouse events in a broader variety of games than the other modes. In addition, some applications and games may have trouble tracking the mouse if it moves too quickly, in which case SetDefaultMouseSpeed can be used to reduce the speed (but only in SendEvent mode). The BlockInput function can be used to prevent any physical mouse activity by the user from disrupting the simulated mouse events produced by the mouse functions. However, this is generally not needed for the SendInput and SendPlay modes because they automatically postpone the user's physical mouse activity until afterward. There is an automatic delay after every click-down and click-up of the mouse (except for SendInput mode and for turning the mouse wheel). Use SetMouseDelay to change the length of the delay. Related Send "{Click}", SendMode, CoordMode, SetDefaultMouseSpeed, SetMouseDelay, MouseClick, MouseClickDrag, MouseMove, ControlClick, BlockInput Examples Clicks the left mouse button at the mouse cursor's current position. Click Clicks the left mouse button at a specific position. Click 100, 200 Moves the mouse cursor to a specific position without clicking. Click 100, 200, 0 Clicks the right mouse button at a specific position. Click 100, 200, "Right" Performs a double-click at the mouse cursor's current position. Click 2 Presses down the left mouse button and holds it. Click "Down" Releases the right mouse button. Click "Up Right" ClipboardAll - Syntax & Usage | AutoHotkey v2 ClipboardAll Creates an object containing everything on the clipboard (such as pictures and formatting). ClipSaved := ClipboardAll(Data, Size) ClipboardAll itself is a class derived from Buffer. Parameters Omit both parameters to retrieve the current contents of the clipboard. Otherwise, specify one or both parameters to create an object containing the given binary clipboard data. Data Type: Object or Integer A Buffer-like object or a pure integer which is the address of the binary data. The data must be in a specific format, so typically originates from a previous call to ClipboardAll. See example #2 below. Size Type: Integer The number of bytes of data to use. This is optional when Data is an object. Return Value Type: Object This function returns a ClipboardAll object, which has two properties (inherited from Buffer): Ptr: The address of the data contained by the object. This address is valid until the object is freed. Size: The size, in bytes, of the raw binary data. Remarks The built-in variable A_Clipboard reflects the current contents of the Windows clipboard expressed as plain text, but can be assigned a ClipboardAll object to restore its content to the clipboard. ClipboardAll is most commonly used to save the clipboard's contents so that the script can temporarily use the clipboard for an operation. When the operation is completed, the script restores the original clipboard contents as shown in example #1 and example #2. If ClipboardAll cannot retrieve one or more of the data objects (formats) on the clipboard, they will be omitted but all the remaining objects will be stored. ClipWait may be used to detect when the clipboard contains data (optionally including non-text data). The binary data contained by the object consists of a four-byte format type, followed by a four-byte data-block size, followed by the data-block for that format. If the clipboard contained more than one format (which is almost always the case), these three items are repeated until all the formats are included. The data ends with a four-byte format type of 0. Known limitation: Using ClipboardAll while cells from Microsoft Excel are on the clipboard may cause Excel to display a "no printers" dialog. Related A_Clipboard, ClipWait, OnClipboardChange, #ClipboardTimeout, Buffer Examples Saves and restores everything on the clipboard using a variable. ClipSaved := ClipboardAll() ; Save the entire clipboard to a variable of your choice. ; ... here make temporary use of the clipboard, such as for quickly pasting large amounts of text ... A_Clipboard := ClipSaved ; Restore the original clipboard. Note the use of A_Clipboard (not ClipboardAll). ClipSaved := "" ; Free the memory in case the clipboard was very large. Saves and restores everything on the clipboard using a file. ; Option 1: Delete any existing file and then use FileAppend. FileDelete "Company Logo.clip" FileAppend ClipboardAll(), "Company Logo.clip" ; The file extension does not matter. ; Option 2: Use FileOpen in overwrite mode and File.RawWrite. ClipData := ClipboardAll() FileOpen("Company Logo.clip", "w").RawWrite(ClipData) To later load the file back onto the clipboard (or into a variable), follow this example: ClipData := FileRead("Company Logo.clip", "RAW") ; In this case, FileRead returns a Buffer. A_Clipboard := ClipboardAll(ClipData) ; Convert the Buffer to a ClipboardAll and assign it. ClipWait - Syntax & Usage | AutoHotkey v2 ClipWait Waits until the clipboard contains data. ClipWait Timeout, WaitForAnyData Parameters Timeout Type: Integer If omitted, the function will wait indefinitely. Otherwise, it will wait no longer than this many seconds (can contain a decimal point). WaitForAnyData Type: Integer If this parameter is omitted or 0 (false), the function is more selective, waiting specifically for text or files to appear ("text" includes anything that would produce text when you paste into Notepad). If this parameter is 1 (true), the function waits for data of any kind to appear on the clipboard. Other values are reserved for future use. Return Value Type: Integer (boolean) This function returns 0 (false) if the function timed out or 1 (true) otherwise (i.e. the clipboard contains data). Remarks It's better to use this function than a loop of your own that checks to see if this clipboard is blank. This is because the clipboard is never opened by this function, and thus it performs better and avoids any chance of interfering with another application that may be using the clipboard. This function considers anything convertible to text (e.g. HTML) to be text. It also considers files, such as those copied in an Explorer window via Ctrl+C, to be text. Such files are automatically converted to their filenames (with full path) whenever the clipboard variable is referred to in the script. See A_Clipboard for details. When 1 (true) is present as the last parameter, the function will be satisfied when any data appears on the clipboard. This can be used in conjunction with ClipboardAll to save non-textual items such as pictures. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. To wait for a fraction of a second, specify a floating point value for the first parameter, for example, 0.25 to wait for a maximum of 250 milliseconds. Related A_Clipboard, WinWait, KeyWait Examples Empties the clipboard, copies the current selection into the clipboard and waits a maximum of 2 seconds until the clipboard contains data. If ClipWait times out, an error message is shown, otherwise the clipboard contents is shown. A_Clipboard := "" ; Empty the clipboard Send "^c" if !ClipWait(2) { MsgBox "The attempt to copy text onto the clipboard failed." return } MsgBox "clipboard = " A_Clipboard return ComCall - Syntax & Usage | AutoHotkey v2 ComCall Calls a native COM interface method by index. Result := ComCall(Index, ComObj , Type1, Arg1, Type2, Arg2, ReturnType) Parameters Index Type: Integer The zero-based index of the method within the virtual function table. Index corresponds to the position of the method within the original interface definition. Microsoft documentation usually lists methods in alphabetical order, which is not relevant. In order to determine the correct index, locate the original interface definition. This may be in a header file or type library. It is important to take into account methods which are inherited from parent interfaces. Since all COM interfaces ultimately derive from IUnknown, the first three methods are always QueryInterface (0), AddRef (1) and Release (2). For example, IShellItem2 is an extension of IShellItem, which starts at index 3 and contains 5 methods, so IShellItem2's first

method (GetPropertyStore) is at index 8. Tip: For COM interfaces defined by Microsoft, try searching the Internet or Windows SDK for "IInterfaceNameVtbl" - for example, "IUnknownVtbl". Microsoft's own interface definitions are accompanied by this plain-C definition of the interface's virtual function table, which lists all methods explicitly, in the correct order. Passing an invalid index may cause undefined behaviour, including (but not limited to) program termination. ComObj Type: Integer, ComValue or Object The target COM object; that is, a COM interface pointer. The pointer value can be passed directly or encapsulated within an object with the Ptr property, such as a ComValue with variant type VT_UNKNOWN. The interface pointer is used to locate the address of the virtual function which implements the interface method, and is also passed as a parameter. This parameter is generally not explicitly present in languages which natively support interfaces, but is shown in the C style "Vtbl" definition. Passing an invalid pointer may cause undefined behaviour, including (but not limited to) program termination. Type1, Arg1 Type: String Each of these pairs represents a single parameter to be passed to the method. The number of pairs is unlimited. For Type, see the DllCall types table. For Arg, specify the value to be passed to the method. ReturnType Type: String If omitted, the return type defaults to HRESULT, which is most the common return type for COM interface methods. Any result indicating failure causes an OSError to be thrown; therefore, the return type must not be omitted unless the actual return type is HRESULT. If the method is of a type that does not return a value (the void return type in C), specify "Int" or any other numeric type without any suffix (except HRESULT), and ignore the return value. As the content of the return value register is arbitrary in such cases, an exception may or may not be thrown if ReturnType is omitted. Otherwise, specify one of the argument types from the DllCall types table. The asterisk suffix is also supported. Although ComCall supports the Cdecl keyword as per DllCall, it is generally not used by COM interface methods. Return Value Type: String or Integer If ReturnType is HRESULT (or omitted) and the method returned an error value (as defined by the FAILED macro), an OSError is thrown. Otherwise, ComCall returns the actual value returned by the method. If the method is of a type that does not return a value (with return type defined in C as void), the result is undefined and should be ignored. Remarks The following DllCall topics are also applicable to ComCall: Types of Arguments and Return Values Errors Native Exceptions and A_LastError Structures and Arrays Known Limitations .NET Framework Related ComObject, ComObjQuery, ComValue, Buffer object, CallbackCreate Examples Remove the active window from the taskbar for 3 seconds. Compare this to the equivalent DllCall example. /* Methods in ITaskbarList's VTable: IUnknown: 0 QueryInterface -- use ComObjQuery instead 1 AddRef -- use ObjAddRef instead 2 Release -- use ObjRelease instead ITaskbarList: 3 HrInit 4 AddTab 5 DeleteTab 6 ActivateTab 7 SetActiveAlt */ IID_ITaskbarList := "{56FDF342-FD6D-11d0-958A-006097C9A090}" CLSID_TaskbarList := "{56FDF344-FD6D-11d0-958A-006097C9A090}" ; Create the TaskbarList object. tbl := ComObject(CLSID_TaskbarList, IID_ITaskbarList) activeHwnd := WinExist("A") ComCall(3, tbl) ; tbl.HrInit() ComCall(5, tbl, "ptr", activeHwnd) ; tbl.DeleteTab(activeHwnd) Sleep 3000 ComCall(4, tbl, "ptr", activeHwnd) ; tbl.AddTab (activeHwnd) ; When finished with the object, simply replace any references with ; some other value (or if its a local variable, just return): tbl := "" Demonstrate some techniques for wrapping COM interfaces. Equivalent to the previous example. tbl := TaskbarList() activeHwnd := WinExist("A") tbl.DeleteTab(activeHwnd) Sleep 3000 tbl.AddTab(activeHwnd) tbl := "" class TaskbarList { static IID := "{56FDF342-FD6D-11d0-958A-006097C9A090}" static CLSID := "{56FDF344-FD6D-11d0-958A-006097C9A090}" ; Called on startup to initialize the class. static __new() { ; Get the base object for all instances of TaskbarList. proto := this.Prototype ; Bound functions can be used to predefine parameters, making ; the methods more usable without requiring wrapper functions. ; HrInit itself has no parameters, so bind only the index, ; and the caller will implicitly provide 'this'. proto.HrInit := ComCall.Bind(3) ; Leave a parameter blank to let the caller provide a value. ; In this case, the blank parameter is 'this' (normally hidden). proto.AddTab := ComCall.Bind(4,, "ptr") ; An object or Map can be used to reduce repetition. for name, args in Map("DeleteTab", [5,,"ptr"], "ActivateTab", [6,,"ptr"], "SetActiveAlt", [7,,"ptr"]) { proto.%name% := ComCall.Bind(args*) } } ; Called by TaskbarList() on the new instance. __new() { this.comobj := ComObject(TaskbarList.CLSID, TaskbarList.IID) this.ptr := this.comobj.ptr ; Request initialization via ITaskbarList. this.HrInit() } } ComObjActive - Syntax & Usage | AutoHotkey v2 ComObjActive Retrieves a registered COM object. ComObj := ComObjActive(CLSID) Parameters CLSID Type: String CLSID or human-readable Prog ID of the COM object to retrieve. Return Value Type: ComObject This function returns a new COM wrapper object with the variant type VT_DISPATCH (9). Error Handling An exception is thrown on failure. Related ComValue, ComObject, ComObjGet, ComObjConnect, ComObjFlags, ObjAddRef/ObjRelease, ComObjQuery, GetActiveObject (Microsoft Docs) Examples Displays the active document in Microsoft Word, if it is running. For details about the COM object and its properties used below, see Word.Application object (Microsoft Docs). word := ComObjActive("Word.Application") if !word MsgBox "Word isn't open." else MsgBox word.ActiveDocument.FullName ComObjArray - Syntax & Usage | AutoHotkey v2 ComObjArray Creates a SafeArray for use with COM. ArrayObj := ComObjArray(VarType, Count1 , Count2, ... Count8) ComObjArray itself is a class derived from ComValue, but is used only to create or identify SafeArray wrapper objects. Parameters VarType Type: Integer The base type of the array (the VARTYPE of each element of the array). The VARTYPE is restricted to a subset of the variant types. Neither the VT_ARRAY nor the VT_BYREF flag can be set. VT_EMPTY and VT_NULL are not valid base types for the array. All other types are legal. See ComObjType for a list of possible values. CountN Type: Integer The size of each dimension. Arrays containing up to 8 dimensions are supported. Return Value Type: ComObjArray This function returns a wrapper object containing a new SafeArray. Methods ComObjArray objects support the following methods: .MaxIndex(n): Returns the upper bound of the nth dimension. If n is omitted, it defaults to 1. .MinIndex(n): Returns the lower bound of the nth dimension. If n is omitted, it defaults to 1. .Clone(): Returns a copy of the array. .__Enum (): Not typically called by script; allows for-loops to be used with SafeArrays. These are currently hard-coded; they do not exist as properties and are not affected by modifications to ComObjArray.Prototype. Remarks ComObjArray objects may also be returned by COM methods and ComValue. Scripts may determine if a value is a ComObjArray as follows: ; Check class if obj is ComObjArray MsgBox "Array subtype: " . ComObjType(obj) & 0xfff else MsgBox "Not an array." ; Check for VT_ARRAY if ComObjType(obj) & 0x2000 MsgBox "obj is a ComObjArray" ; Check specific array type if ComObjType(obj) = 0x2008 MsgBox "obj is a ComObjArray of strings" Arrays with up to 8 dimensions are supported. Since SafeArrays are not designed to support multiple references, when one SafeArray is assigned to an element of another SafeArray, a separate copy is created. However, this only occurs if the wrapper object has the F_OWNVALUE flag, which indicates it is responsible for destroying the array. This flag can be removed by using ComObjFlags. When a function or method called by a COM client returns a SafeArray with the F_OWNVALUE flag, a copy is created and returned instead, as the original SafeArray is automatically destroyed. Related ComValue, ComObjType, ComObjValue, ComObjActive, ComObjFlags, Array Manipulation Functions (Microsoft Docs) Examples Simple usage. arr := ComObjArray (VT_VARIANT:=12, 3) arr[0] := "Auto" arr[1] := "Hot" arr[2] := "key" t := "" Loop arr.MaxIndex() + 1 t .= arr[A_Index-1] MsgBox t Multiple dimensions. arr := ComObjArray(VT_VARIANT:=12, 3, 4) ; Get the number of dimensions: dim := DllCall("oleaut32\SafeArrayGetDim", "ptr", ComObjValue(arr)) ; Get the bounds of each dimension: dims := "" Loop dim dims .= arr.MinIndex(A_Index) " .. " arr.MaxIndex(A_Index) "`n" MsgBox dims ; Simple usage: Loop 3 { x := A_Index-1 Loop 4 { y := A_Index-1 arr[x, y] := x * y } } MsgBox arr[2, 3] ComObjConnect - Syntax & Usage | AutoHotkey v2 ComObjConnect Connects a COM object's event source to the script, enabling events to be handled. ComObjConnect ComObj , PrefixOrSink Parameters ComObj Type: ComObject An object which raises events. If the object does not support the IConnectionPointContainer interface or type information about the object's class cannot be retrieved, an error message is shown. This can be suppressed or handled with try/catch. The IProvideClassInfo interface is used to retrieve type information about the object's class if the object supports it. Otherwise, ComObjConnect attempts to retrieve type information via the object's IDispatch interface, which may be unreliable. PrefixOrSink Type: String or Object A string to prefix to the event name to determine which global function to call when an event occurs, or an event sink object defining a static method for each event to be handled. Note: Nested functions are not supported in this mode, as names may be resolved after the current function returns. To use nested functions or closures, attach them to an object and pass the object as described below. If omitted, the object is "disconnected"; that is, the script will no longer receive notification of its events. Usage To make effective use of ComObjConnect, you must first write functions in the script to handle any events of interest. Such functions, or "event-handlers," have the following structure: PrefixEventName([Params..., ComObj]) { ... event-handling code ... return ReturnValue } Prefix should be the same as the PrefixOrSink parameter if it is a string; otherwise, it should be omitted. EventName should be replaced with the name of whatever event the function should handle. Params corresponds to whatever parameters the event has. If the event has no parameters, Params should be omitted entirely. ComObj is an additional parameter containing a reference to the original wrapper object which was passed to ComObjConnect; it is never included in the COM event's documentation. "ComObj" should be replaced with a name more meaningful in the context of your script. Note that event handlers may have return values. To return a COM-specific type of value, use ComValue. For example, return ComValue(0,0) returns a variant of type VT_EMPTY, which is equivalent to returning undefined (or not returning) from a JavaScript function. Call ComObjConnect(yourObject, "Prefix") to enable event-handling. Call ComObjConnect(yourObject) to disconnect the object (stop handling events). If the number of parameters is not known, a variadic function can be used. Event Sink If PrefixOrSink is an object, whenever an event is raised, the corresponding method of that object is called. Although the object can be constructed dynamically, it is more typical for PrefixOrSink to refer to a class or an instance of a class. In that case, methods are defined as shown above, but without Prefix. As with any call to a method, the method's (normally hidden) this parameter contains a reference to the object through which the method was called; i.e. the event sink object, not the COM object. This can be used to provide context to the event handlers, or share values between them. To catch all events without defining a method for each one, define a __Call meta-function. ComObject releases its reference to PrefixOrSink automatically if the COM object releases the connection. For example, Internet Explorer does this when it exits. If the script does not retain its own reference to PrefixOrSink, it can use __Delete to detect when this occurs. If the object is hosted by a remote process and the process terminates unexpectedly, it may take several minutes for the system to release the connection. Remarks The script must retain a reference to ComObj, otherwise it would be freed automatically and would disconnect from its COM object, preventing any further events from being detected. There is no standard way to detect when the connection is no longer required, so the script must disconnect manually by calling ComObjConnect. The Persistent function may be needed to keep the script running while it is listening for events. An exception is thrown on failure. Related ComObject, ComObjGet, ComObjActive, WScript.ConnectObject (Microsoft Docs) Examples Launches an instance of Internet Explorer and connects events to corresponding

script functions with the prefix "IE_". For details about the COM object and DocumentComplete event used below, see InternetExplorer object (Microsoft Docs). ie := ComObject("InternetExplorer.Application") ; Connects events to corresponding script functions with the prefix "IE_". ComObjConnect(ie, "IE_") ie.Visible := true ; This is known to work incorrectly on IE7. ie.Navigate("https://www.autohotkey.com/") Persistent IE_DocumentComplete(ieEventParam, &url, ieFinalParam) { ; IE passes url as a reference to a VARIANT, therefore &url is used above ; so that the code below can refer to it naturally rather than as %url%. s := "" if (ie != ieEventParam) s .= "First parameter is a new wrapper object.`n" if (ie == ieFinalParam) s .= "Final parameter is the original wrapper object.`n" if (ComObjValue(ieEventParam) == ComObjValue(ieFinalParam)) s .= "Both wrapper objects refer to the same IDispatch instance.`n" MsgBox s . "Finished loading " ie.Document.title " @ " url ie.Quit() ExitApp } ComObject - Syntax & Usage | AutoHotkey v2 ComObject Creates a COM object. ComObj := ComObject (CLSID , IID) ComObject itself is a class derived from ComValue, but is used only to create or identify COM objects. Parameters CLSID Type: String CLSID or human-readable Prog ID of the COM object to create. IID Type: String The identifier of the interface to return. In most cases this is omitted; if omitted, it defaults to "{00020400-0000-0000-C000-000000000046}" (IID_IDispatch). Return Value Type: ComValue or ComObject This function returns a COM wrapper object of type dependent on the IID parameter. IIDClassVariant TypeDescription IID_IDispatch ComObject VT_DISPATCH (9) Allows the script to call properties and methods of the object using normal object syntax. Any other IID ComValue VT_UNKNOWN (13) Provides only a Ptr property, which allows the object to be passed to DllCall or ComCall. Error Handling An exception is thrown on failure, such as if a parameter is invalid or the object does not support the interface specified by IID. Related ComValue, ComObjGet, ComObjActive, ComObjConnect, ComObjArray, ComObjQuery, ComCall, CreateObject (Microsoft Docs) Examples For a long list of examples, see the following forum topic: https://www.autohotkey.com/forum/topic61509.html. Launches an instance of Internet Explorer, makes it visible and navigates to a website. ie := ComObject("InternetExplorer.Application") ie.Visible := true ; This is known to work incorrectly on IE7. ie.Navigate("https://www.autohotkey.com/") Retrieve the path of the desktop's current wallpaper. AD_GETWP_BMP := 0 AD_GETWP_LAST_APPLIED := 0x00000002 CLSID_ActiveDesktop := "{75048700-EF1F-11D0-9888-006097DEACF9}" IID_IActiveDesktop := "{F490EB00-1240-11D1-9888-006097DEACF9}" cchWallpaper := 260 GetWallpaper := 4 AD := ComObject(CLSID_ActiveDesktop, IID_IActiveDesktop) wszWallpaper := Buffer(cchWallpaper * 2) ComCall (GetWallpaper, AD, "ptr", wszWallpaper, "uint", cchWallpaper, "uint", AD_GETWP_LAST_APPLIED) Wallpaper := StrGet(wszWallpaper, "UTF-16") MsgBox "Wallpaper: " Wallpaper ComObjFlags - Syntax & Usage | AutoHotkey v2 ComObjFlags Retrieves or changes flags which control a COM wrapper object's behaviour. Flags := ComObjFlags(ComObj , NewFlags, Mask) Parameters ComObj Type: ComValue A COM wrapper object. NewFlags Type: Integer New values for the flags identified by Mask, or flags to add or remove. Mask Type: Integer A bitmask of flags to change. Return Value Type: Integer This function returns the current flags of the specified COM object (after applying NewFlags, if specified). Error Handling A TypeError is thrown if ComObj is not a COM wrapper object. Flags Flag Effect 1 F_OWNVALUE SafeArray: If the flag is set, the SafeArray is destroyed when the wrapper object is freed. Since SafeArrays have no reference counting mechanism, if a SafeArray with this flag is assigned to an element of another SafeArray, a separate copy is created. BSTR: If the flag is set, the BSTR is freed when the wrapper object is freed. The flag is set automatically when a BSTR is allocated as a result of type conversion performed by ComValue, such as ComValue(8, "example"). Remarks If Mask is omitted, NewFlags specifies the flags to add (if positive) or remove (if negative). For example, ComObjFlags(obj, -1) removes the F_OWNVALUE flag. Do not specify any value for Mask other than 0 or 1; all other bits are reserved for future use. Related ComValue, ComObjActive, ComObjArray Examples Checks for the presence of the F_OWNVALUE flag. arr := ComObjArray(0xC, 1) if ComObjFlags(arr) & 1 MsgBox "arr will be automatically destroyed." else MsgBox "arr will not be automatically destroyed." Changes array-in-array behaviour. arr1 := ComObjArray (0xC, 3) arr2 := ComObjArray(0xC, 1) arr2[0] := "original value" arr1[0] := arr2 ; Assign implicit copy. ComObjFlags(arr2, -1) ; Remove F_OWNVALUE. arr1 [1] := arr2 ; Assign original array. arr1[2] := arr2.Clone() ; Assign explicit copy. arr2[0] := "new value" for arr in arr1 MsgBox arr[0] arr1 := "" ; Not valid since arr2 == arr1[1], which has been destroyed: ; arr2[0] := "foo" ComObjFromPtr - Syntax & Usage | AutoHotkey v2 ComObjFromPtr Wraps a raw IDispatch pointer (COM object) for use by the script. ComObj := ComObjFromPtr(DispPtr) Parameters DispPtr Type: Integer A non-null interface pointer for IDispatch or a derived interface. Return Value Type: ComObject Returns a wrapper object containing the variant type VT_DISPATCH and the given pointer. Wrapping a COM object enables the script to interact with it more naturally, using object syntax. However, the majority of scripts do not need to do this manually since a wrapper object is created automatically by ComObject, ComObjActive, ComObjGet and any COM method which returns an object. Remarks The wrapper object assumes responsibility for automatically releasing the pointer when appropriate. This function queries the object for its IDispatch interface; if one is returned, DispPtr is immediately released. Therefore, if the script intends to use the pointer after calling this function, it must call ObjAddRef(DispPtr) first. Known limitation: Each time a COM object is wrapped, a new wrapper object is created. Comparisons and assignments such as obj1 == obj2 and array[obj1] := value treat the two wrapper objects as unique, even when they contain the same COM object. Related ComObject, ComValue, ComObjGet, ComObjConnect, ComObjFlags, ObjAddRef/ObjRelease, ComObjQuery, GetActiveObject (Microsoft Docs) ComObjGet - Syntax & Usage | AutoHotkey v2 ComObjGet Returns a reference to an object provided by a COM component. ComObj := ComObjGet(Name) Parameters Name Type: String The display name of the object to be retrieved. See MkParseDisplayName (Microsoft Docs) for more information. Return Value Type: ComObject This function returns a new COM wrapper object with the variant type VT_DISPATCH (9). Error Handling An exception is thrown on failure. Related ComObject, ComObjActive, ComObjConnect, ComObjQuery, CoGetObject (Microsoft Docs) Examples Press Shift+Esc to show the command line which was used to launch the active window's process. For Win32_Process, see Microsoft Docs. +Esc:: { pid := WinGetPID("A") ; Get WMI service object. wmi := ComObjGet("winmgmts:") ; Run query to retrieve matching process(es). queryEnum := wmi.ExecQuery("" . "Select * from Win32_Process where ProcessId=" . pid) ._NewEnum() ; Get first matching process. if queryEnum(&proc) MsgBox (proc.CommandLine, "Command line", 0) else MsgBox("Process not found!") } ComObjQuery - Syntax & Usage | AutoHotkey v2 ComObjQuery Queries a COM object for an interface or service. InterfaceComObj := ComObjQuery(ComObj, SID, IID) Parameters ComObj Type: ComValue, Object or Integer A COM wrapper object, an interface pointer, or an object with a Ptr property which returns an interface pointer. IID Type: String An interface identifier (GUID) in the form "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". SID Type: String A service identifier in the same form as IID. When omitting this parameter, also omit the comma. Return Value Type: ComValue or ComObject This function returns a COM wrapper object of type dependent on the IID parameter. IIDClassVariant TypeDescription IID_IDispatch ComObject VT_DISPATCH (9) Allows the script to call properties and methods of the object using normal object syntax. Any other IID ComValue VT_UNKNOWN (13) Provides only a Ptr property, which allows the object to be passed to DllCall or ComCall. Error Handling An exception is thrown on failure, such as if the interface is not supported. Remarks In its two-parameter mode, this function is equivalent to IUnknown::QueryInterface. When SID and IID are both specified, it internally queries for the IServiceProvider interface, then calls IServiceProvider::QueryService. ComCall can be used to call native interface methods. Related ComCall, ComObject, ComObjGet, ComObjActive Examples Determines the class name of an object. obj := ComObject ("Scripting.Dictionary") MsgBox "Interface name: " ComObjType(obj, "name") IID_IProvideClassInfo := "{B196B283-BAB4-101A-B69C-00AA00341D07}" ; Request the object's IProvideClassInfo interface. try pci := ComObjQuery(obj, IID_IProvideClassInfo) catch { MsgBox "IProvideClassInfo interface not supported." return } ; Call GetClassInfo to retrieve a pointer to the ITypeInfo interface. ComCall(3, pci, "ptr*", &ti := 0) ; Wrap ti to ensure automatic cleanup. ti := ComValue(13, ti) ; Call GetDocumentation to get the object's full type name. ComCall(12, ti, "int", -1, "ptr*", &pname := 0, "ptr", 0, "ptr", 0, "ptr", 0) ; Convert the BSTR pointer to a usable string. name := StrGet(pname, "UTF-16") ; Clean up. DllCall("oleaut32\SysFreeString", "ptr", pname) pci := ti := "" ; Display the type name! MsgBox "Class name: " name Automates an existing Internet Explorer window. sURL := "https://www.autohotkey.com/boards/" if WebBrowser := GetWebBrowser() WebBrowser.Navigate(sURL) GetWebBrowser() { ; Get a raw pointer to the document object of the top-most IE window. static msg := DllCall("RegisterWindowMessage", "Str", "WM_HTML_GETOBJECT") lResult := SendMessage(msg, 0, 0, "Internet Explorer_Server1", "ahk_class IEFrame") if !lResult return ; IE not found. static IID_IHTMLDocument2 := GUID("{332C4425-26CB-11D0-B483-00C04FD90119}") static VT_UNKNOWN := 13 DllCall("oleacc\ObjectFromLresult", "Ptr", lResult , "Ptr", IID_IHTMLDocument2, "Ptr", 0 , "Ptr*", pdoc := ComValue(VT_UNKNOWN, 0)) ; Query for the WebBrowserApp service. In this particular case, ; the SID and IID are the same, but it isn't always this way. static IID_IWebBrowserApp := "{0002DF05-0000-0000-C000-000000000046}" static SID_SWebBrowserApp := IID_IWebBrowserApp pweb := ComObjQuery(pdoc, SID_SWebBrowserApp, IID_IWebBrowserApp) ; Return the WebBrowser object as IDispatch for usability. ; This works only because IWebBrowserApp is derived from IDispatch. ; pweb will release its ptr automatically, so AddRef to counter that. ObjAddRef(pweb.ptr) static VT_DISPATCH := 9 return ComValue(VT_DISPATCH, pweb.ptr) } GUID(sGUID) ; Converts a string to a binary GUID and returns it in a Buffer. { GUID := Buffer(16, 0) if DllCall("ole32\CLSIDFromString", "WStr", sGUID, "Ptr", GUID) < 0 throw ValueError("Invalid parameter #1", -1, sGUID) return GUID } ComObjType - Syntax & Usage | AutoHotkey v2 ComObjType Retrieves type information from a COM object. VarType := ComObjType(ComObj) IName := ComObjType(ComObj, "Name") IID := ComObjType(ComObj, "IID") CName := ComObjType(ComObj, "Class") CLSID := ComObjType(ComObj, "CLSID") Parameters ComObj Type: ComValue A wrapper object containing a COM object or typed value. Param2 Type: String The second parameter is a string indicating the type information to retrieve. Return Value The return value depends on the value of Param2: Param2 Return Type Return Value Omitted Integer A variant type code which indicates the type of value contained by the COM wrapper object. "Name" String The name of the object's default interface. "IID" String The globally unique identifier (GUID) of the object's default interface. "Class" String The object's class name. Note that this is not the same as a Prog ID (a Prog ID is a name used to identify the class in the system registry, or for ComObject). "CLSID" String The globally unique identifier (GUID) of the object's class. Classes are often registered by CLSID under the HKCR\CLSID registry key. An empty string is returned if either parameter is invalid or if the requested type information could not be retrieved. Variant Type Constants COM uses the following values to identify basic data types: VT_EMPTY := 0 ; No value VT_NULL := 1 ; SQL-style Null VT_I2 := 2 ; 16-bit signed int VT_I4 := 3 ; 32-bit signed int VT_R4 := 4 ; 32-bit floating-point number VT_R8 := 5 ; 64-bit floating-point number VT_CY := 6 ; Currency VT_DATE := 7 ; Date VT_BSTR := 8 ; COM

string (Unicode string with length prefix) VT_DISPATCH := 9 ; COM object VT_ERROR := 0xA ; Error code (32-bit integer) VT_BOOL := 0xB ; Boolean True (-1) or False (0) VT_VARIANT := 0xC ; VARIANT (must be combined with VT_ARRAY or VT_BYREF) VT_UNKNOWN := 0xD ; IUnknown interface pointer VT_DECIMAL := 0xE ; (not supported) VT_I1 := 0x10 ; 8-bit signed int VT_UI1 := 0x11 ; 8-bit unsigned int VT_UI2 := 0x12 ; 16-bit unsigned int VT_UI4 := 0x13 ; 32-bit unsigned int VT_I8 := 0x14 ; 64-bit signed int VT_UI8 := 0x15 ; 64-bit unsigned int VT_INT := 0x16 ; Signed machine int VT_UINT := 0x17 ; Unsigned machine int VT_RECORD := 0x24 ; User-defined type -- NOT SUPPORTED VT_ARRAY := 0x2000 ; SAFEARRAY VT_BYREF := 0x4000 ; Pointer to another type of value /* VT_ARRAY and VT_BYREF are combined with another value (using bitwise OR) to specify the exact type. For instance, 0x2003 identifies a SAFEARRAY of 32-bit signed integers and 0x400C identifies a pointer to a VARIANT. */ General Remarks In most common cases, return values from methods or properties of COM objects are converted to an appropriate data type supported by AutoHotkey. Types which aren't specifically handled are coerced to strings via VariantChangeType; if this fails or if the variant type contains the VT_ARRAY or VT_BYREF flag, an object containing both the value and its type is returned instead. For any variable x, if ComObjType(x) returns an integer, x contains a COM object wrapper. If Param2 is "Name" or "IID", type information is retrieved via the IDispatch::GetTypeInfo interface method. ComObj's variant type must be VT_DISPATCH. If Param2 is "Class" or "CLSID", type information is retrieved via the IProvideClassInfo::GetClassInfo interface method. ComObj's variant type must be VT_DISPATCH or VT_UNKNOWN, and the object must implement the IProvideClassInfo interface (some objects do not). Related ComObjValue, ComValue, ComObject, ComObjGet, ComObjActive Examples Reports various type information of a COM object. d := ComObject("Scripting.Dictionary") MsgBox ("Variant type:`t" ComObjType(d) " Interface name:`t" ComObjType(d, "Name") " Interface ID:`t" ComObjType(d, "IID") " Class name:`t" ComObjType(d, "Class") " Class ID (CLSID):`t" ComObjType(d, "CLSID")) ComObjValue - Syntax & Usage | AutoHotkey v2 ComObjValue Retrieves the value or pointer stored in a COM wrapper object. Value := ComObjValue(ComObj) Parameters ComObj Type: ComValue A wrapper object containing a COM object or typed value. Return Value Type: Integer This function returns a 64-bit signed integer. Error Handling A TypeError is thrown if ComObj is not a COM wrapper object. Remarks This function is not intended for general use. Calling ComObjValue is equivalent to variant.llVal, where ComObj is treated as a VARIANT structure. Any script which uses this function must be aware what type of value the wrapper object contains and how it should be treated. For instance, if an interface pointer is returned, Release should not be called, but AddRef may be required depending on what the script does with the pointer. Related ComObjType, ComObject, ComObjGet, ComObjActive ComValue - Syntax & Usage | AutoHotkey v2 ComValue Wraps a value, SafeArray or COM object for use by the script or for passing to a COM method. ComObj := ComValue(VarType, Value , Flags) ComValue itself is a class derived from Any, but is used only to create or identify COM wrapper objects. Parameters VarType Type: Integer An integer indicating the type of value. See ComObjType for a list of types. Value The value to wrap. If this is a pure integer and VarType is not VT_R4, VT_R8, VT_DATE or VT_CY, its value is used directly; in particular, VT_BSTR, VT_DISPATCH and VT_UNKNOWN can be initialized with a pointer value. In any other case, the value is copied into a temporary VARIANT using the same rules as normal COM methods calls. If the source variant type is not equal to VarType, conversion is attempted by calling VariantChangeType with a wFlags value of 0. An exception is thrown if conversion fails. Flags Type: Integer Flags affecting the behaviour of the wrapper object; see ComObjFlags for details. Return Value Type: ComValue, ComValueRef, ComObjArray or ComObject Returns a wrapper object containing a variant type and value or pointer. This object has multiple uses: Some COM methods may require specific types of values which have no direct equivalent within AutoHotkey. This function allows the type of a value to be specified when passing it to a COM method. For example, ComValue(0xB, true) creates an object which represents the COM boolean value true. Wrapping a COM object or SafeArray enables the script to interact with it more naturally, using object syntax. However, the majority of scripts do not need to do this manually since a wrapper object is created automatically by ComObject, ComObjArray, ComObjActive, ComObjGet and any COM method which returns an object. Wrapping a COM interface pointer allows the script to take advantage of automatic reference counting. An interface pointer can be wrapped immediately upon being returned to the script (typically from ComCall or DllCall), avoiding the need to explicitly release it at some later point. Ptr If a wrapper object's VarType is VT_UNKNOWN (13) or includes the VT_BYREF (0x4000) flag or VT_ARRAY (0x2000) flag, the Ptr property can be used to retrieve the address of the object, typed variable or SafeArray. This allows the ComObject itself to be passed to any DllCall or ComCall parameter which has the "Ptr" type, but can also be used explicitly. For example, ComObj.Ptr is equivalent to ComObjValue(ComObj) in these cases. If a wrapper object's VarType is VT_UNKNOWN (13) or VT_DISPATCH (9) and the wrapped pointer is null (0), the Ptr property can be used to retrieve the current null value or to assign a pointer to the wrapper object. Once assigned (if non-null), the pointer will be released automatically when the wrapper object is freed. This can be used with DllCall or ComCall output parameters of type "Ptr*" or "PtrP" to ensure the pointer will be released automatically, such as if an error occurs. For an example, see ComObjQuery. When a wrapper object with VarType VT_DISPATCH (9) and a null (0) pointer value is assigned a non-null pointer value, its type changes from ComValue to ComObject. The properties and methods of the wrapped object become available and the Ptr property becomes unavailable. ByRef If a wrapper object's VarType includes the VT_BYREF (0x4000) flag, empty brackets [] can be used to read or write the referenced value. When creating a reference, Value must be the memory address of a variable or buffer with sufficient capacity to store a value of the given type. For example, the following can be used to create a variable which a VBScript function can write into: vbuf := Buffer(24, 0) vref := ComValue(0x400C, vbuf.ptr) ; 0x400C is a combination of VT_BYREF and VT_VARIANT. vref[] := "in value" sc.Run("Example", vref) ; sc should be initialized as in the example below. MsgBox vref[] Note that although any previous value is freed when a new value is assigned by vref[] or the COM method, the final value is not freed automatically. Freeing the value requires knowing which type it is. Because it is VT_VARIANT in this case, it can be freed by calling VariantClear with DllCall or by using a simpler method: assign an integer, such as vref[] := 0. If the method accepts a combination of VT_BYREF and VT_VARIANT as shown above, a VarRef can be used instead. For example: some_var := "in value" sc.Run("Example", &some_var) MsgBox some_var However, some methods require a more specific variant type, such as VT_BYREF | VT_I4. In such cases, the first approach shown above must be used, replacing 0x400C with the appropriate variant type. General Remarks When this function is used to wrap an IDispatch or IUnknown interface pointer (passed as an integer), the wrapper object assumes responsibility for automatically releasing the pointer when appropriate. Therefore, if the script intends to use the pointer after calling this function, it must call ObjAddRef(DispPtr) first. By contrast, this is not necessary if Value is itself a ComValue or ComObject. Conversion from VT_UNKNOWN to VT_DISPATCH results in a call to IUnknown::QueryInterface, which may produce an interface pointer different to the original, and will throw an exception if the object does not implement IDispatch. By contrast, if Value is an integer and VarType is VT_DISPATCH, the value is used directly, and therefore must be an IDispatch-compatible interface pointer. The VarType of a wrapper object can be retrieved using ComObjType. The Value of a wrapper object can be retrieved using ComObjValue. Known limitation: Each time a COM object is wrapped, a new wrapper object is created. Comparisons and assignments such as obj1 == obj2 and array[obj1] := value treat the two wrapper objects as unique, even when they contain the same variant type and value. Related ComObjFromPtr, ComObject, ComObjGet, ComObjConnect, ComObjFlags, ObjAddRef/ObjRelease, ComObjQuery, GetActiveObject (Microsoft Docs) Examples Passes a VARIANT ByRef to a COM function. ; Preamble - ScriptControl requires a 32-bit version of AutoHotkey. code := " (Sub Example(Var) MsgBox Var Var = "out value!" End Sub)" sc := ComObject("ScriptControl"), sc.Language := "VBScript", sc.AddCode(code) ; Example: Pass a VARIANT ByRef to a COM method. var := ComVar() var[] := "in value" sc.Run("Example", var.ref) MsgBox var[] ; The same thing again, but more direct: variant_buf := Buffer(24, 0) ; Make a buffer big enough for a VARIANT. var := ComValue(0x400C, variant_buf.ptr) ; Make a reference to a VARIANT. var[] := "in value" sc.Run("Example", var) ; Pass the VT_BYREF ComValue itself, no [] or .ref. MsgBox var[] ; If a VARIANT contains a string or object, it must be explicitly freed ; by calling VariantClear or assigning a pure numeric value: var[] := 0 ; The simplest way when the method accepts VT_BYREF|VT_VARIANT: var := "in value" sc.Run("Example", &var) MsgBox var ; ComVar: An object which can be used to pass a value ByRef. ; this[] retrieves the value. ; this[] := Val sets the value. ; this.ref retrieves a ByRef object for passing to a COM method. class ComVar { __new (vType := 0xC) { ; Allocate memory for a VARIANT to hold our value. VARIANT is used even ; when vType != VT_VARIANT so that VariantClear can be used by __delete. this.var := Buffer(24, 0) ; Create an object which can be used to pass the variable ByRef. this.ref := ComValue(0x4000|vType, this.var.ptr + (vType=0xC ? 0 : 8)) ; Store the variant type for VariantClear (if not VT_VARIANT). if Type != 0xC NumPut "ushort", vType, this.var } __item { get => this.ref[] set => this.ref[] := value } __delete() { DllCall("oleaut32\VariantClear", "ptr", this.var) } } Continue - Syntax & Usage | AutoHotkey v2 Continue Skips the rest of a loop statement's current iteration and begins a new one. Continue LoopLabel Parameters LoopLabel LoopLabel identifies which loop this statement should apply to; either by label name or numeric nesting level. If omitted or 1, this statement applies to the innermost loop in which it is enclosed. If a label is specified, it must point directly at a loop statement. LoopLabel must be a constant value - variables and expressions are not supported, with the exception of a single literal number or quoted string enclosed in parentheses. For example: continue("outer") Remarks Continue behaves the same as reaching the loop's closing brace: It increases A_Index by 1. It skips the rest of the loop's body. The loop's condition (if it has one) is checked to see if it is satisified. If so, a new iteration begins; otherwise the loop ends. The use of Break and Continue are encouraged over Goto since they usually make scripts more readable and maintainable. Related Break, Loop, Until, While-loop, For-loop, Blocks, Labels Examples Displays 5 message boxes, one for each number between 6 and 10. Note that in the first 5 iterations of the loop, the Continue statement causes the loop to start over before it reaches the MsgBox line. Loop 10 { if (A_Index <= 5) continue MsgBox A_Index } Continues the outer loop from within a nested loop. outer: Loop 3 { x := A_Index Loop 3 { if (x*A_Index = 4) continue outer ; Equivalent to continue 2 or goto continue_outer. MsgBox x "," A_Index } continue_outer: ; For goto. } List of Control Functions | AutoHotkey v2 Control Functions Functions to retrieve information about a control, or make a variety of changes to a control. Click on a function name for details. Function Description ControlAddItem Adds the specified string as a new entry at the bottom of a ListBox or ComboBox. ControlChooseIndex Sets the selection in a ListBox, ComboBox or Tab control to be the specified entry or tab number. ControlChooseString Sets the selection in a ListBox or ComboBox to be the first entry whose leading part matches the specified string. ControlClick Sends a mouse button or mouse wheel event to a control. ControlDeleteItem Deletes the specified entry number from a ListBox or ComboBox. ControlFindItem Returns the entry

number of a ListBox or ComboBox that is a complete match for the specified string. ControlFocus Sets input focus to a given control on a window. ControlGetChecked Returns a non-zero value if the checkbox or radio button is checked. ControlGetChoice Returns the name of the currently selected entry in a ListBox or ComboBox. ControlGetClassNN Returns the ClassNN (class name and sequence number) of the specified control. ControlGetEnabled Returns a non-zero value if the specified control is enabled. ControlGetFocus Retrieves which control of the target window has keyboard focus, if any. ControlGetHwnd Returns the unique ID number of the specified control. ControlGetIndex Returns the index of the currently selected entry or tab in a ListBox, ComboBox or Tab control. ControlGetItems Returns an array of items/rows from a ListBox, ComboBox, or DropDownList. ControlGetPos Retrieves the position and size of a control. ControlGetStyleControlGetExStyle Returns an integer representing the style or extended style of the specified control. ControlGetText Retrieves text from a control. ControlGetVisible Returns a non-zero value if the specified control is visible. ControlHide Hides the specified control. ControlHideDropDown Hides the drop-down list of a ComboBox control. ControlMove Moves or resizes a control. ControlSendControlSendText Sends simulated keystrokes or text to a window or control. ControlSetChecked Turns on (checks) or turns off (unchecks) a checkbox or radio button. ControlSetEnabled Enables or disables the specified control. ControlSetStyleControlSetExStyle Changes the style or extended style of the specified control, respectively. ControlSetText Changes the text of a control. ControlShow Shows the specified control if it was previously hidden. ControlShowDropDown Shows the drop-down list of a ComboBox control. EditGetCurrentCol Returns the column number in an Edit control where the caret (text insertion point) resides. EditGetCurrentLine Returns the line number in an Edit control where the caret (text insert point) resides. EditGetLine Returns the text of the specified line in an Edit control. EditGetLineCount Returns the number of lines in an Edit control. EditGetSelectedText Returns the selected text in an Edit control. EditPaste Pastes the specified string at the caret (text insertion point) in an Edit control. ListViewGetContent Returns a list of items/rows from a ListView. The Control Parameter Functions which operate on individual controls have a parameter named Control which supports a few different ways to identify the control. The Control parameter can be one of the following: ClassNN (String): The ClassNN (classname and instance number) of the control, which can be determined via Window Spy. For example "Edit1" is the first control with classname "Edit". Text (String): The control's text. The matching behavior is determined by SetTitleMatchMode. HWND (Integer): The control's HWND, which is typically retrieved via ControlGetHwnd, MouseGetPos, or DllCall. This also works on hidden controls even when DetectHiddenWindows is Off. Any subsequent window parameters are ignored. Object: An object of any type with a Hwnd property, such as a GuiControl. A PropertyError is thrown if the object has no Hwnd property, or TypeError if it does not return a pure integer. Any subsequent window parameters are ignored. Omitted: A few functions are able to operate on either a control or a top-level window. Omitting the Control parameter causes the function to use the target window (specified by WinTitle) instead of one of its controls. For example, ControlSend can send keyboard messages directly to the window. Error Handling Typically one of the following errors may be thrown: TargetError: The target window or control could not be found. Error or OSError: There was a problem carring out the function's purpose, such as retrieving a setting or applying a change. ValueError or TypeError: Invalid parameters were detected. Remarks To improve reliability, a delay is done automatically after each use of a Control function that changes a control (except for ControlSetStyle and ControlSetExStyle). That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. To discover the ClassNN or HWND of the control that the mouse is currently hovering over, use MouseGetPos. To retrieve a list of all controls in a window, use WinGetControls or WinGetControlsHwnd. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SetControlDelay, Win functions, GuiControl object (for controls created by the script) ControlAddItem - Syntax & Usage | AutoHotkey v2 ControlAddItem Adds the specified string as a new entry at the bottom of a ListBox or ComboBox. ControlAddItem String, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters String Type: String The string to add. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the index of the new item, where 1 is the first item, 2 is the second, etc. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error or OSError is thrown if the item could not be added. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlDeleteItem, ControlFindItem, Add method (GuiControl object), Control functions ControlChooseIndex - Syntax & Usage | AutoHotkey v2 ControlChooseIndex Sets the selection in a ListBox, ComboBox or Tab control to be the specified entry or tab number. ControlChooseIndex N, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters N Type: Integer The index of the entry or tab, where 1 is the first entry or tab, 2 is the second, etc. To deselect all entries in a ListBox or ComboBox, specify 0. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo", "List" or "Tab". An Error or OSError is thrown if the change could not be applied. Remarks To select all items in a multi-select listbox, follow this example: PostMessage(0x0185, 1, -1, "ListBox1", WinTitle) ; Select all listbox items. 0x0185 is LB_SETSEL. To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetIndex, ControlChooseString, Choose method (GuiControl object), Control functions ControlChooseString - Syntax & Usage | AutoHotkey v2 ControlChooseString Sets the selection in a ListBox or ComboBox to be the first entry whose leading part matches the specified string. ControlChooseString String, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters String Type: String The string to choose. The search is not case sensitive. For example, if a ListBox/ComboBox contains the item "UNIX Text", specifying the word "unix" (lowercase) would be enough to select it. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the index of the chosen item, where 1 is the first item, 2 is the second, etc. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error or OSError is thrown if the change could not be applied. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlChooseIndex, ControlGetChoice, Choose method (GuiControl object), Control functions ControlClick - Syntax & Usage | AutoHotkey v2 ControlClick Sends a mouse button or mouse wheel event to a control. ControlClick Control-or-Pos, WinTitle, WinText, WhichButton, ClickCount, Options, ExcludeTitle, ExcludeText Parameters Control-or-Pos Type: String, Integer or Object If this parameter is omitted, the target window itself will be clicked. Otherwise, one of the two modes below will be used. Mode 1 (Position): Specify the X and Y coordinates relative to the upper left corner of the target window's client area. The X coordinate must precede the Y coordinate and there must be at least one space or tab between them. For example: X55 Y33. If there is a control at the specified coordinates, it will be sent the click-event at those exact coordinates. If there is no control, the target window itself will be sent the event (which might have no effect depending on the nature of the window). Note: In mode 1, the X and Y option letters of the Options parameter are ignored. Mode 2 (Control): Specify the control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. By default, mode 2 takes precedence over mode 1. For example, in the unlikely event that there is a control whose text or ClassNN has the format "Xnnn Ynnn", it would be acted upon by Mode 2. To override this and use mode 1 unconditionally, specify the word Pos in Options as in the following example: ControlClick "x255 y152", WinTitle,,,, "Pos". WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. WhichButton Type: String The button to click: LEFT, RIGHT, MIDDLE (or just the first letter of each of these). If omitted or blank, the LEFT button will be used. X1 (XButton1, the 4th mouse button) and X2 (XButton2, the 5th mouse button) are also supported. WheelUp (or WU), WheelDown (or WD), WheelLeft (or WL) and WheelRight (or WR) are also supported. In this case, ClickCount is the number of notches to turn the wheel. ClickCount Type: Integer The number of clicks to send. If omitted or blank, 1 click is sent. Options Type: String A series of zero or more of the following option letters. For example: d x50 y25 NA: May improve reliability. See reliability below. D: Press the mouse button down but do not release it (i.e. generate a down-event). If both the D and U options are absent, a complete click (down and up) will be sent. U: Release the mouse button (i.e. generate an up-event). This option should not be present if the D option is already present (and vice versa). Pos: Specify the word Pos anywhere in Options to unconditionally use the X/Y positioning mode as described in the Control-or-Pos parameter above. Xn: Specify for n the X position to click at, relative to the control's upper left corner. If unspecified, the click will occur at the

horizontal-center of the control. Yn: Specify for n the Y position to click at, relative to the control's upper left corner. If unspecified, the click will occur at the vertical-center of the control. Use decimal (not hexadecimal) numbers for the X and Y options. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling An exception is thrown in the following cases: TargetError: The target window could not be found. TargetError: The target control could not be found and Control-or-Pos does not specify a valid position. OSError (very rare): the X or Y position is omitted and the control's position could not be determined. ValueError or TypeError: Invalid parameters were detected. Reliability To improve reliability -- especially during times when the user is physically moving the mouse during the ControlClick -- one or both of the following may help: 1) Use SetControlDelay -1 prior to ControlClick. This avoids holding the mouse button down during the click, which in turn reduces interference from the user's physical movement of the mouse. 2) Specify the string NA anywhere in the sixth parameter (Options) as shown below: SetControlDelay -1 ControlClick "Toolbar321", WinTitle,,,, "NA" NA avoids marking the target window as active and avoids merging its input processing with that of the script, which may prevent physical movement of the mouse from interfering (but usually only when the target window is not active). However, this method might not work for all types of windows and controls. Remarks Not all applications obey a ClickCount higher than 1 for turning the mouse wheel. For those applications, use a Loop to turn the wheel more than one notch as in this example, which turns it 5 notches: Loop 5 ControlClick Control, WinTitle, WinText, "WheelUp" Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SetControlDelay, Control functions, Click Examples Clicks the OK button. ControlClick "OK", "Some Window Title" Clicks at a set of coordinates. Note the lack of a comma between X and Y. ControlClick "x55 y77", "Some Window Title" Clicks in NA mode at coordinates that are relative to a named control. SetControlDelay -1 ; May improve reliability and reduce side effects. ControlClick "Toolbar321", "Some Window Title",,,, "NA x192 y10" ControlDeleteItem - Syntax & Usage | AutoHotkey v2 ControlDeleteItem Deletes the specified entry number from a ListBox or ComboBox. ControlDeleteItem N, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters N Type: Integer The index of the item, where 1 is the first entry, 2 is the second, etc. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error or OSError is thrown if the item could not be deleted. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlAddItem, ControlFindItem, Delete method (GuiControl object), Control functions ControlFindItem - Syntax & Usage | AutoHotkey v2 ControlFindItem Returns the entry number of a ListBox or ComboBox that is a complete match for the specified string. FoundItem := ControlFindItem(String, Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters String Type: String The string to find. The search is case-insensitive. Unlike ControlChooseString, the entry's entire text must match, not just the leading part. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the entry number of a ListBox or ComboBox that is a complete match for String. The first entry in the control is 1, the second 2, and so on. If no match is found, an exception is thrown. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error is thrown if the item could not be found. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlAddItem, ControlDeleteItem, Control functions ControlFocus - Syntax & Usage | AutoHotkey v2 ControlFocus Sets input focus to a given control on a window. ControlFocus Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks To be effective, the control's window generally must not be minimized or hidden. To improve reliability, a delay is done automatically after every use of this function. That delay can be changed via SetControlDelay. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. When a control is focused in response to user input (such as pressing the Tab key), the dialog manager applies additional effects which are independent of the control having focus. These effects are not applied by ControlFocus, and therefore the following limitations apply: Focusing a button does not automatically make it the default button, as would normally happen if a button is focused by user input. The default button can usually be activated by pressing Enter. If user input previously caused the default button to be temporarily changed, focusing a non-button control does not automatically restore the default highlight to the actual default button. Pressing Enter may then activate the default button even though it is not highlighted. Focusing an edit control does not automatically select its text. Instead, the insertion point (caret) is typically positioned wherever it was last time the control had focus. The WM_NEXTDLGCTL message can be used to focus the control and apply these additional effects. For example: WinExist("A") ; Set the Last Found Window to the active window ControlGet, hWndControl, Hwnd,, Button1 ; Get HWND of first Button SendMessage, 0x0028, hWndControl, True ; 0x0028 is WM_NEXTDLGCTL Related SetControlDelay, ControlGetFocus, Control functions Examples Sets the input focus to the OK button. ControlFocus "OK", "Some Window Title" ; Set focus to the OK button ControlGetChecked - Syntax & Usage | AutoHotkey v2 ControlGetChecked Returns a non-zero value if the checkbox or radio button is checked. IsChecked := ControlGetChecked(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer (boolean) This function returns 1 (true) if the checkbox or radio button is checked, or 0 (false) if not. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetChecked, Value property (GuiControl object), Control functions ControlGetChoice - Syntax & Usage | AutoHotkey v2 ControlGetChoice Returns the name of the currently selected entry in a ListBox or ComboBox. Choice := ControlGetChoice(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the name of the currently selected entry in a ListBox or ComboBox. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error is thrown on failure. Remarks To instead retrieve the position of the selected item, follow this example (use only one of the first two lines): ChoicePos := SendMessage(0x0188, 0, 0, "ListBox1", WinTitle) ; 0x0188 is LB_GETCURSEL (for a ListBox). ChoicePos := SendMessage (0x0147, 0, 0, "ComboBox1", WinTitle) ; 0x0147 is CB_GETCURSEL (for a DropDownList or ComboBox). ChoicePos += 1 ; Convert from 0-based to 1-based, i.e. so that the first item is known as 1, not 0. ; ChoicePos is now 0 if there is no item selected. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlChooseIndex, ControlChooseString, Value property (GuiControl object), Choose method (GuiControl object), Control functions ControlGetClassNN - Syntax & Usage | AutoHotkey v2 ControlGetClassNN Returns the ClassNN (class name and sequence number) of the specified control. ClassNN := ControlGetClassNN(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the ClassNN (class name and sequence number) of the specified control. Error Handling A TargetError is thrown if there is a problem determining the target window or control. An Error or OSError is thrown if the ClassNN could not be determined.

Remarks A control's ClassNN is the name of its window class followed by its sequence number within the top-level window which contains it. For example, "Edit1" is the first Edit control on a window and "Button12" is the twelth button. A control's ClassNN can also be determined via Window Spy, MouseGetPos or WinGetControls. Some class names include digits which are not part of the control's sequence number. For example, "SysListView321" is the window's first ListView control, not its 321st. To retrieve the class name without the sequence number, pass the control's HWND to WinGetClass. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetClass, WinGetControls, ClassNN property (GuiControl object), MouseGetPos, Control functions Examples Retrieves the ClassNN of the currently focused control. classNN := ControlGetClassNN (ControlGetFocus("A")) ControlGetEnabled - Syntax & Usage | AutoHotkey v2 ControlGetEnabled Returns a non-zero value if the specified control is enabled. IsEnabled := ControlGetEnabled(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer (boolean) This function returns 1 (true) if the specified control is enabled, or 0 (false) if disabled. Error Handling A TargetError is thrown if the window or control could not be found. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetEnabled, WinSetEnabled, Enabled property (GuiControl object), Control functions ControlGetFocus - Syntax & Usage | AutoHotkey v2 ControlGetFocus Retrieves which control of the target window has keyboard focus, if any. HWND := ControlGetFocus(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the window handle (HWND) of the focused control. If none of the target window's controls has focus, the return value is 0. Error Handling A TargetError is thrown if there is a problem determining the target window or control. An OSError is thrown if there is a problem determining the focus. Remarks The control retrieved by this function is the one that has keyboard focus, that is, the one that would receive keystrokes if the user were to type any. The target window must be active to have a focused control, but even the active window may lack a focused control. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlFocus, Control functions Examples Reports the HWND and ClassNN of the active window's focused control. FocusedHwnd := ControlGetFocus("A") FocusedClassNN := ControlGetClassNN(FocusedHwnd) MsgBox 'Control with focus = {Hwnd: ' FocusedHwnd ', ClassNN: "' FocusedClassNN '"}' ControlGetHwnd - Syntax & Usage | AutoHotkey v2 ControlGetHwnd Returns the unique ID number of the specified control. Hwnd := ControlGetHwnd(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the window handle (HWND) of the specified control. Error Handling A TargetError is thrown if the window or control could not be found. Remarks A control's HWND is often used with PostMessage, SendMessage, and DllCall. On a related note, a control's HWND can also be retrieved via MouseGetPos. Finally, a control's HWND can be used directly in a WinTitle parameter. This also works on hidden controls even when DetectHiddenWindows is Off. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetID, Hwnd property (GuiControl object), Control functions Examples Retrieves the unique ID number of the Notepad's Edit control. editHwnd := ControlGetHwnd("Edit1", "ahk_class Notepad") ControlGetIndex - Syntax & Usage | AutoHotkey v2 ControlGetIndex Returns the index of the currently selected entry or tab in a ListBox, ComboBox or Tab control. Index := ControlGetIndex(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the index of the currently selected entry or tab. The first entry or tab is 1, the second is 2, etc. If no entry or tab is selected, the return value is 0. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo", "List" or "Tab". An OSError is thrown if a message could not be sent to the control. Remarks To instead discover how many tabs (pages) exist in a tab control, follow this example: TabCount := SendMessage(0x1304,,, "SysTabControl321", WinTitle) ; 0x1304 is TCM_GETITEMCOUNT. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlChooseIndex, ControlGetChoice, ControlChooseString, Value property (GuiControl object), Choose method (GuiControl object), Control functions Examples Retrieves the active tab number of the first Tab control. WhichTab := ControlGetIndex("SysTabControl321", "Some Window Title") MsgBox "Tab #" WhichTab " is active." ControlGetItems - Syntax & Usage | AutoHotkey v2 ControlGetItems Returns an array of items/rows from a ListBox, ComboBox, or DropDownList. Items := ControlGetItems(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Array of Strings This function returns an array containing the text of each item or row. Error Handling A TargetError is thrown if the window or control could not be found, or if the control's class name does not contain "Combo" or "List". An Error is thrown on failure, such as if a message returned a failure code or could not be sent. Remarks Some applications store their item data privately, which prevents their text from being retrieved. In these cases, an exception will usually not be thrown, but all the retrieved fields will be empty. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ListViewGetContent, WinGetList, Control functions Examples Accesses the items one by one. for item in ControlGetItems("ComboBox1", WinTitle) MsgBox "Item number " A_Index " is " item Accesses a specific item by index. items := ControlGetItems("ListBox1", WinTitle) MsgBox "The first item is " items[1] MsgBox "The last item is " items[-1] ControlGetPos - Syntax & Usage | AutoHotkey v2 ControlGetPos Retrieves the position and size of a control. ControlGetPos &OutX, &OutY, &OutWidth, &OutHeight, Control, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters &OutX, &OutY Type: VarRef References to the output variables in which to store the X and Y coordinates (in pixels) of Control's upper left corner. These coordinates are relative to the upper-left corner of the target window's client area and thus are the same as those used by ControlMove. If either X or Y is omitted, the corresponding values will not be stored. &OutWidth, &OutHeight Type: VarRef References to the output variables in which to store Control's width and height (in pixels). If omitted, the corresponding values will not be stored. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks Unlike functions that change a control, ControlGetPos does not have an automatic delay (SetControlDelay does not affect it). Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlMove, WinGetPos, Control functions Examples Continuously updates and displays the name and position of the control currently under the mouse cursor. Loop { Sleep 100 MouseGetPos ,, &WhichWindow, &WhichControl try ControlGetPos &x, &y, &w, &h, WhichControl, WhichWindow ToolTip WhichControl "`nX" X "`tY" Y "`nW" W "`t" H } ControlGetStyle / ControlGetExStyle - Syntax & Usage | AutoHotkey v2 ControlGetStyle / ControlGetExStyle Returns an integer representing the style or extended style of the specified control. Style := ControlGetStyle(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) ExStyle := ControlGetExStyle(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer These functions return the style or extended style of the specified control. Error Handling A TargetError is thrown if the window or control could not be found. Remarks See the styles table for a partial listing of styles. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetStyle /

ControlSetExStyle, WinGetStyle / WinGetExStyle, styles table, Control functions ControlGetText - Syntax & Usage | AutoHotkey v2 ControlGetText Retrieves text from a control. Text := ControlGetText(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the text of the specified control. Error Handling A TargetError is thrown if the window or control could not be found. Remarks Note: To retrieve text from a ListView, ListBox, or ComboBox, use ListViewGetContent or ControlGetItems instead. If the retrieved text appears to be truncated (incomplete), it may be necessary to retrieve the text by sending the WM_GETTEXT message via SendMessage instead. This is because some applications do not respond properly to the WM_GETTEXTLENGTH message, which causes AutoHotkey to make the return value too small to fit all the text. This function might use a large amount of RAM if the target control (e.g. an editor with a large document open) contains a large quantity of text. However, a variable's memory can be freed after use by assigning it to nothing, i.e. Text := "". Text retrieved from most control types uses carriage return and linefeed (`r`n) rather than a solitary linefeed (`n) to mark the end of each line. It is not necessary to do SetTitleMatchMode "Slow" because ControlGetText always retrieves the text using the slow method (since it works on a broader range of control types). To retrieve a list of all controls in a window, use WinGetControls or WinGetControlsHwnd. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetText, WinGetText, Control functions Examples Retrieves the current text from Notepad's edit control and stores it in Text. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. Text := ControlGetText("Edit1", "Untitled -") Retrieves and reports the current text from the main window's edit control. ListVars WinWaitActive "ahk_class AutoHotkey" MsgBox ControlGetText("Edit1") ; Use the window found above. ControlGetVisible - Syntax & Usage | AutoHotkey v2 ControlGetVisible Returns a non-zero value if the specified control is visible. IsVisible := ControlGetVisible(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer (boolean) This function returns 1 (true) if the specified control is visible, or 0 (false) if hidden. Error Handling A TargetError is thrown if the window or control could not be found. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlHide, ControlShow, Visible property (GuiControl object), Control functions ControlHide - Syntax & Usage | AutoHotkey v2 ControlHide Hides the specified control. ControlHide Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks If you additionally want to prevent a control's shortcut key (underlined letter) from working, disable the control via ControlSetEnabled. To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlShow, ControlGetVisible, WinHide, Visible property (GuiControl object), Control functions ControlHideDropDown - Syntax & Usage | AutoHotkey v2 ControlHideDropDown Hides the drop-down list of a ComboBox control. ControlHideDropDown Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control. Related ControlShowDropDown, Control functions Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Examples See example #1 on the ControlShowDropDown page. ControlMove - Syntax & Usage | AutoHotkey v2 ControlMove Moves or resizes a control. ControlMove X, Y, Width, Height, Control, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters X, Y Type: Integer The X and Y coordinates (in pixels) of the upper left corner of Control's new location. If either coordinate is omitted, Control's position in that dimension will not be changed. The coordinates are relative to the upper-left corner of the target window's client area; ControlGetPos can be used to determine them. Width, Height Type: Integer The new width and height of Control (in pixels). If either parameter is omitted, Control's size in that dimension will not be changed. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if the control's current position could not be determined. Remarks To improve reliability, a delay is done automatically after every use of this function. That delay can be changed via SetControlDelay. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetPos, WinMove, SetControlDelay, Control functions Examples Demonstrates how to manipulate the OK button of an input box while the script is waiting for user input. SetTimer ControlMoveTimer IB := InputBox(, "My Input Box") ControlMoveTimer() { if !WinExist("My Input Box") return ; Otherwise the above set the "last found" window for us: SetTimer , 0 WinActivate ControlMove 10,, 200,, "OK" ; Move the OK button to the left and increase its width. } ControlSend[Text] - Syntax & Usage | AutoHotkey v2 ControlSend / ControlSendText Sends simulated keystrokes or text to a window or control. ControlSend Keys , Control, WinTitle, WinText, ExcludeTitle, ExcludeText ControlSendText: Same parameters as above. Parameters Keys Type: String The sequence of keys to send (see the Send function for details). The rate at which characters are sent is determined by SetKeyDelay. Unlike the Send function, mouse clicks cannot be sent by ControlSend. Use ControlClick for that. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. If this parameter is omitted, the keystrokes will be sent directly to the target window instead of one of its controls (see Automating Winamp for an example). WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks ControlSendText sends the individual characters of the Keys parameter without translating {Enter} to Enter, ^c to Ctrl+C, etc. For details, see Text mode. It is also valid to use {Raw} or {Text} with ControlSend. If the Control parameter is omitted, this function will attempt to send directly to the target window by sending to its topmost control (which is often the correct one) or the window itself if there are no controls. This is useful if a window does not appear to have any controls at all, or just for the convenience of not having to worry about which control to send to. By default, modifier keystrokes (Ctrl, Alt, Shift, and Win) are sent as they normally would be by the Send function. This allows command prompt and other console windows to properly detect uppercase letters, control characters, etc. It may also improve reliability in other ways. However, in some cases these modifier events may interfere with the active window, especially if the user is actively typing during a ControlSend or if Alt is being sent (since Alt activates the active window's menu bar). This can be avoided by explicitly sending modifier up and down events as in this example: ControlSend "{Alt down}f{Alt up}", "Edit1", "Untitled - Notepad" The method above also allows the sending of modifier keystrokes (Ctrl, Alt, Shift, and Win) while the workstation is locked (protected by logon prompt). BlockInput should be avoided when using ControlSend against a console window such as command prompt. This is because it might prevent capitalization and modifier keys such as Ctrl from working properly. The value of SetKeyDelay determines the speed at which keys are sent. If the target window does not receive the keystrokes reliably, try increasing the press duration via the second parameter of SetKeyDelay as in these examples: SetKeyDelay 10, 10 SetKeyDelay 0, 10 SetKeyDelay -1, 0 If the target control is an Edit control (or something similar), the following are usually more reliable and faster than ControlSend: EditPaste("This text will be inserted at the caret position.", ControlName, WinTitle) ControlSetText("This text will entirely replace any current text.", ControlName, WinTitle) ControlSend is generally not capable of manipulating a window's menu bar. To work around this, use MenuSelect. If that is not possible due to the nature of the menu bar, you could try to discover the message that corresponds to the desired menu item by following the SendMessage Tutorial. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SetKeyDelay, Escape

sequences (e.g. `n) , Control functions, Send, Automating Winamp Examples Opens Notepad minimized and send it some text. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. Run "Notepad"„ "Min", &PID ; Run Notepad minimized. WinWait "ahk_pid " PID ; Wait for it to appear. ; Send the text to the inactive Notepad edit control. ; The third parameter is omitted so the last found window is used. ControlSend "This is a line of text in the notepad window.{Enter}", "Edit1" ControlSendText "Notice that {Enter} is not sent as an Enter keystroke with ControlSendText.", "Edit1" Msgbox "Press OK to activate the window to see the result." WinActivate "ahk_pid " PID ; Show the result. Opens the command prompt and sent it some text. This example may fail on Windows 11 or later, as it requires the classic version of the command prompt. SetTitleMatchMode 2 Run A_ComSpec„ &PID ; Run command prompt. WinWait "ahk_pid " PID ; Wait for it to appear. ControlSend "ipconfig{Enter}"„ "cmd.exe" ; Send directly to the command prompt window. Creates a GUI with an edit control and sent it some text. MyGui := Gui() MyGui.Add("Edit", "r10 w500") MyGui.Show() ControlSend "This is a line of text in the edit control. {Enter}", "Edit1", MyGui ControlSendText "Notice that {Enter} is not sent as an Enter keystroke with ControlSendText.", "Edit1", MyGui ControlSetChecked - Syntax & Usage | AutoHotkey v2 ControlSetChecked Turns on (checks) or turns off (unchecks) a checkbox or radio button. ControlSetChecked Value, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Value Type: Integer One of the following values: 1 or True turns on the setting 0 or False turns off the setting -1 sets it to the opposite of its current state Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control. Remarks To ensure correct functionality, this function also sets the input focus to the control. To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetChecked, Value property (GuiControl object), Control functions ControlSetEnabled - Syntax & Usage | AutoHotkey v2 ControlSetEnabled Enables or disables the specified control. ControlSetEnabled Value, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Value Type: Integer One of the following values: 1 or True turns on the setting 0 or False turns off the setting -1 sets it to the opposite of its current state Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetEnabled, WinSetEnabled, Enabled property (GuiControl object), Control functions ControlSetStyle / ControlSetExStyle - Syntax & Usage | AutoHotkey v2 ControlSetStyle / ControlSetExStyle Changes the style or extended style of the specified control, respectively. ControlSetStyle Value, Control , WinTitle, WinText, ExcludeTitle, ExcludeText ControlSetExStyle Value, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Value Type: Integer or String Pass a positive integer to completely overwrite the window's style; that is, to set it to Value. To easily add, remove or toggle styles, pass a numeric string prefixed with a plus sign (+), minus sign (-) or caret (^), respectively. The new style value is calculated as shown below (where CurrentStyle could be retrieved with ControlGetStyle/ControlGetExStyle or WinGetStyle/WinGetExStyle): Operation Prefix Example String Formula Add + +0x80 NewStyle := CurrentStyle | Value Remove - -0x80 NewStyle := CurrentStyle & ~Value Toggle ^ ^0x80 NewStyle := CurrentStyle ^ Value If Value is a negative integer, it is treated the same as the corresponding numeric string. To use the + or ^ prefix literally in an expression, the prefix or value must be enclosed in quote marks. For example: WinSetStyle("+0x80") or WinSetStyle("^" StylesToToggle). This is because the expression +123 produces 123 (without a prefix) and ^123 is a syntax error. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if the style could not be changed. Partial change is considered a success. Remarks See the styles table for a partial listing of styles. Certain style changes require that the entire window be redrawn using WinRedraw. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetStyle / ControlGetExStyle, WinSetStyle / WinSetExStyle, styles table, Control functions Examples Sets the WS_BORDER style of the Notepad's Edit control to its opposite state. ControlSetStyle("^0x800000", "Edit1", "ahk_class Notepad") ControlSetText - Syntax & Usage | AutoHotkey v2 ControlSetText Changes the text of a control. ControlSetText NewText, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters NewText Type: String The new text to set into the control. If blank or omitted, the control is made blank. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks Most control types use carriage return and linefeed (`r`n) rather than a solitary linefeed (`n) to mark the end of each line. To translate a block of text containing `n characters, follow this example: MyVar := StrReplace(MyVar, "`n", "`r`n") To improve reliability, a delay is done automatically after every use of this function. That delay can be changed via SetControlDelay. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SetControlDelay, ControlGetText, Control functions Examples Changes the text of Notepad's edit control. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. ControlSetText("New Text Here", "Edit1", "Untitled -") Changes the text of the main window's edit control. ListVars WinWaitActive "ahk_class AutoHotkey" ControlSetText "New Text Here", "Edit1" ; Use the window found above. ControlShow - Syntax & Usage | AutoHotkey v2 ControlShow Shows the specified control if it was previously hidden. ControlShow Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlHide, ControlGetVisible, WinShow, Visible property (GuiControl object), Control functions ControlShowDropDown - Syntax & Usage | AutoHotkey v2 ControlShowDropDown Shows the drop-down list of a ComboBox control. ControlShowDropDown Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control. Remarks To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlHideDropDown, Control functions Examples Opens the Run dialog, shows its drop-down list for 2 seconds and closes the dialog. Send "#r" ; Open the Run dialog. WinWaitActive "ahk_class #32770" ; Wait for the dialog to appear. ControlShowDropDown "ComboBox1" ; Show the drop-down list. The second parameter is omitted so that the last found window is used. Sleep 2000 ControlHideDropDown "ComboBox1" ; Hide the drop-down list. Sleep 1000 Send "{Esc}" ; Close the Run dialog. CoordMode - Syntax & Usage | AutoHotkey v2 CoordMode Sets coordinate mode for various built-in functions to be relative to either the active window or the screen. CoordMode TargetType , RelativeTo Parameters TargetType Type: String The type of target to affect. Specify one of the following words: ToolTip: Affects ToolTip. Pixel: Affects PixelGetColor, PixelSearch, and ImageSearch. Mouse: Affects MouseGetPos, Click, and MouseMove/Click/Drag. Caret: Affects CaretGetPos. Menu: Affects the Menu.Show method when coordinates are specified for it. RelativeTo Type: String The area to which TargetType is to be related. Specify one of the following words (if omitted, it defaults to Screen): Screen: Coordinates are relative to the desktop (entire screen). Window:

Coordinates are relative to the active window. Client: Coordinates are relative to the active window's client area, which excludes the window's title bar, menu (if it has a standard one) and borders. Client coordinates are less dependent on OS version and theme. Return Value Type: String This function returns the previous setting. Remarks If this function is not used, all built-in functions except those documented otherwise (e.g. WinMove and InputBox) use coordinates that are relative to the active window's client area. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. The built-in A_CoordMode variables contain the current settings. Related Click, MouseMove, MouseClick, MouseClickDrag, MouseGetPos, PixelGetColor, PixelSearch, ToolTip, Menu.Show Examples Places tooltips at absolute screen coordinates. CoordMode "ToolTip", "Screen" Same effect as the above because "Screen" is the default. CoordMode "ToolTip" Critical - Syntax & Usage | AutoHotkey v2 Critical Prevents the current thread from being interrupted by other threads, or enables it to be interrupted. Critical OnOffNumeric Parameters OnOffNumeric Type: String or Integer If blank or omitted, it defaults to On. Otherwise, specify one of the following: On: The current thread is made critical, meaning that it cannot be interrupted by another thread. Off: The current thread immediately becomes interruptible, regardless of the settings of Thread Interrupt. See Critical Off for details. (Numeric): Specify a positive number to turn on Critical but also change the number of milliseconds between checks of the internal message queue. See Message Check Interval for details. Specifying 0 turns off Critical. Specifying -1 turns on Critical but disables message checks. Return Value Type: Integer This function returns the previous setting (the value A_IsCritical would return prior to calling the function). Behavior of Critical Threads Critical threads are uninterruptible; for details, see Threads. A critical thread becomes interruptible when a message box or other dialog is displayed. However, unlike Thread Interrupt, the thread becomes critical again after the user dismisses the dialog. Critical Off When buffered events are waiting to start new threads, using Critical "Off" will not result in an immediate interruption of the current thread. Instead, an average of 5 milliseconds will pass before an interruption occurs. This makes it more than 99.999% likely that at least one line after Critical "Off" will execute before an interruption. You can force interruptions to occur immediately by means of a delay such as a Sleep -1 or a WinWait for a window that does not yet exist. Critical "Off" cancels the current thread's period of uninterruptibility even if the thread was not Critical, thereby letting events such as Size be processed sooner or more predictably. Thread Settings See A_IsCritical for how to save and restore the current setting of Critical. However, since Critical is a thread-specific setting, when a critical thread ends, the underlying/resumed thread (if any) will be automatically noncritical. Consequently there is no need to do Critical "Off" right before ending a thread. If Critical is not used by the auto-execute thread, all threads start off as noncritical (though the settings of Thread "Interrupt" will still apply). By contrast, if the auto-execute thread turns on Critical but never turns it off, every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off critical. The function Thread "NoTimers" is similar to Critical except that it only prevents interruptions from timers. Message Check Interval Specifying a positive number as the first parameter (e.g. Critical 30) turns on Critical but also changes the minimum number of milliseconds between checks of the internal message queue. If unspecified, the default is 16 milliseconds while Critical is On, and 5 ms while Critical is Off. Increasing the interval postpones the arrival of messages/events, which gives the current thread more time to finish. This reduces the possibility that certain OnMessage() and GUI events will be lost due to "thread already running". However, functions that wait such as Sleep and WinWait will check messages regardless of this setting (a workaround is DllCall("Sleep", "UInt", 500)). This setting affects the following: Preemptive message checks, which potentially occur before each line of code executes. Periodic message checks during Send, file and download operations. It does not affect the frequency of message checks while the script is paused or waiting. Because the system tick count generally has a granularity of 15.6 milliseconds, the minimum delta value is generally at least 15 or 16. The duration since the last check must exceed the interval setting in order for another check to occur. For example, a setting of 16 requires the tick count to change by 17 or greater. As a message check could occur at any time within the 15.6 millisecond window, any setting between 1 and 16 could permit two message checks within a single interval. Note: Increasing the message-check interval too much may reduce the responsiveness of various events such as GUI window repainting. Critical -1 turns on Critical but disables message checks. This does not prevent the program from checking for messages while performing a sleep, delay or wait. It is useful in cases where dispatching messages could interfere with the code currently executing, such as while handling certain types of messages during an OnMessage callback. Related Thread (function), Threads, #MaxThreadsPerHotkey, #MaxThreadsBuffer, OnMessage, CallbackCreate, Hotkey, Menu object, SetTimer Examples Press a hotkey to display a tooltip for 3 seconds. Due to Critical, any new thread that is launched during this time (e.g. by pressing the hotkey again) will be postponed until the tooltip disappears. #space:: ; Win+Space hotkey. { Critical ToolTip "No new threads will launch until after this ToolTip disappears." Sleep 3000 ToolTip ; Turn off the tip. return ; Returning from a hotkey function ends the thread. Any underlying thread to be resumed is noncritical by definition. } DateAdd - Syntax & Usage | AutoHotkey v2 DateAdd Adds or subtracts time from a date-time value. Result := DateAdd(DateTime, Time, TimeUnits) Parameters DateTime Type: String A date-time stamp in the YYYYMMDDHH24MISS format. Time Type: Integer or Float The amount of time to add, as an integer or floating-point number. Specify a negative number to perform subtraction. TimeUnits Type: String The meaning of the Time parameter. TimeUnits may be one of the following strings (or just the first letter): Seconds, Minutes, Hours or Days. Return Value Type: String This function returns a string of digits in the YYYYMMDDHH24MISS format. This string should not be treated as a number (one should not perform math on it or compare it numerically). Remarks The built-in variable A_Now contains the current local time in YYYYMMDDHH24MISS format. To calculate the amount of time between two timestamps, use DateDiff. If DateTime contains an invalid timestamp or a year prior to 1601, a ValueError is thrown. Related DateDiff, FileGetTime, FormatTime Examples Calculates the date 31 days from now. later := DateAdd(A_Now, 31, "days") MsgBox FormatTime(later) DateDiff - Syntax & Usage | AutoHotkey v2 DateDiff Compares two date-time values and returns the difference. Result := DateDiff(DateTime1, DateTime2, TimeUnits) Parameters DateTime1 DateTime2 Type: String Date-time stamps in the YYYYMMDDHH24MISS format. If DateTime1 is earlier than DateTime2, the result is a negative number. If either parameter is an empty string, the current local date and time (A_Now) is used. TimeUnits Type: String Units to measure the difference in. TimeUnits may be one of the following strings (or just the first letter): Seconds, Minutes, Hours or Days. Return Value Type: Integer This function returns the difference between the two timestamps, in the units specified by TimeUnits. The result is always rounded down to the nearest integer. For example, if the actual difference between two timestamps is 1.999 days, it will be reported as 1 day. If higher precision is needed, specify Seconds for TimeUnits and divide the result by 60.0, 3600.0, or 86400.0. Remarks The built-in variable A_Now contains the current local time in YYYYMMDDHH24MISS format. To precisely determine the elapsed time between two events, use the A_TickCount method because it provides millisecond precision. To add or subtract a certain number of seconds, minutes, hours, or days from a timestamp, use DateAdd (subtraction is achieved by adding a negative number). If DateTime contains an invalid timestamp or a year prior to 1601, a ValueError is thrown. Related DateAdd, FileGetTime, FormatTime Examples Calculates the number of days between two timestamps and reports the result. var1 := "20050126" var2 := "20040126" MsgBox DateDiff(var1, var2, "days") ; The answer will be 366 since 2004 is a leap year. DetectHiddenText - Syntax & Usage | AutoHotkey v2 DetectHiddenText Determines whether invisible text in a window is "seen" for the purpose of finding the window. This affects built-in functions such as WinExist and WinActivate. DetectHiddenText Mode Parameters Mode Type: Boolean 1 or True: Hidden text is detected. This is the default. 0 or False: Hidden text is not detected. Return Value Type: Boolean This function returns the previous setting. Remarks "Hidden text" is a term that refers to those controls of a window that are not visible. Their text is thus considered "hidden". Turning off DetectHiddenText can be useful in cases where you want to detect the difference between the different panes of a multi-pane window or multi-tabbed dialog. Use Window Spy to determine which text of the currently-active window is hidden. All built-in functions that accept a WinText parameter are affected by this setting, including WinActivate, WinActive, WinWait, and WinExist. The built-in variable A_DetectHiddenText contains the current setting (1 or 0). Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related DetectHiddenWindows Examples Turns off the detection of hidden text. DetectHiddenText False DetectHiddenWindows - Syntax & Usage | AutoHotkey v2 DetectHiddenWindows Determines whether invisible windows are "seen" by the script. DetectHiddenWindows Mode Parameters Mode Type: Boolean 1 or True: Hidden windows are detected. 0 or False: This is the default. Hidden windows are not detected, except by the WinShow function. Return Value Type: Boolean This function returns the previous setting. Remarks Turning on DetectHiddenWindows can make scripting harder in some cases since some hidden system windows might accidentally match the title or text of another window you're trying to work with. So most scripts should leave this setting turned off. However, turning it on may be useful if you wish to work with hidden windows directly without first using WinShow to unhide them. All windowing functions except WinShow are affected by this setting, including WinActivate, WinActive, WinWait and WinExist. By contrast, WinShow will always unhide a hidden window even if hidden windows are not being detected. Turning on DetectHiddenWindows is not necessary in the following cases: When using a pure HWND, as in WinShow(A_ScriptHwnd). When using an object with a Hwnd property (such as a Gui), as in WinMoveTop(myGui). When accessing a control or child window via the ahk_id method or as the last-found-window. When accessing GUI windows via the +LastFound option. Cloaked windows are also considered hidden. Cloaked windows, introduced with Windows 8, are windows on a non-active virtual desktop or UWP apps which have been suspended to improve performance, or more precisely to reduce their memory consumption. On Windows 10, the processes of those are indicated with a green leaf in the Task Manager. Such windows are hidden from view, but might still have the WS_VISIBLE window style. The built-in variable A_DetectHiddenWindows contains the current setting (1 or 0). Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related DetectHiddenText Examples Turns on the detection of hidden windows. DetectHiddenWindows True DirCopy - Syntax & Usage | AutoHotkey v2 DirCopy Copies a folder along with all its sub-folders and files (similar to xcopy). DirCopy Source, Dest , Overwrite Parameters Source Type: String Name of the source directory (with no trailing backslash), which is assumed to be in A_WorkingDir if an absolute path isn't specified. For example: C:\My Folder If supported by the OS, Source can also be the path of a zip file, in which case its content will be copied to the destination directory. This has been confirmed to work on Windows 7 and Windows 11. Dest Type: String Name of the destination directory (with no trailing

baskslash), which is assumed to be in A_WorkingDir if an absolute path isn't specified. For example: C:\Copy of My Folder Overwrite Type: Integer This parameter determines whether to overwrite files if they already exist. If omitted, it defaults to 0 (false). Specify one of the following values: 0 (false): Do not overwrite existing files. The operation will fail and have no effect if Dest already exists as a file or directory. 1 (true): Overwrite existing files. However, any files or subfolders inside Dest that do not have a counterpart in Source will not be deleted. Other values are reserved for future use. Error Handling An exception is thrown if an error occurs. If the source directory contains any saved webpages consisting of a PageName.htm file and a corresponding directory named PageName_files, an exception may be thrown even when the copy is successful. Remarks If the destination directory structure doesn't exist it will be created if possible. Since the operation will recursively copy a folder along with all its subfolders and files, the result of copying a folder to a destination somewhere inside itself is undefined. To work around this, first copy it to a destination outside itself, then use DirMove to move that copy to the desired location. DirCopy copies a single folder. To instead copy the contents of a folder (all its files and subfolders), see the examples section of FileCopy. Related DirMove, FileCopy, FileMove, FileDelete, file-loops, DirSelect, SplitPath Examples Copies a directory to a new location. DirCopy "C:\My Folder", "C:\Copy of My Folder" Prompts the user to copy a folder. SourceFolder := DirSelect(, 3, "Select the folder to copy") if SourceFolder = "" return ; Otherwise, continue. TargetFolder := DirSelect(, 3, "Select the folder IN WHICH to create the duplicate folder.") if TargetFolder = "" return ; Otherwise, continue. Result := MsgBox("A copy of the folder '" SourceFolder "' will be put into '" TargetFolder "'. Continue?",, 4) if Result = "No" return SplitPath SourceFolder, &SourceFolderName ; Extract only the folder name from its full path. try DirCopy SourceFolder, TargetFolder "\" SourceFolderName catch MsgBox "The folder could not be copied, perhaps because a folder of that name already exists in '" TargetFolder "'." return DirCreate - Syntax & Usage | AutoHotkey v2 DirCreate Creates a folder. DirCreate DirName Parameters DirName Type: String Name of the directory to create, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Error Handling An OSError is thrown if an error occurs. A_LastError is set to the result of the operating system's GetLastError() function. Remarks This function will also create all parent directories given in DirName if they do not already exist. Related DirDelete Examples Creates a new directory, including its parent directories if necessary. DirCreate "C:\Test1\My Images\Folder2" DirDelete - Syntax & Usage | AutoHotkey v2 DirDelete Deletes a folder. DirDelete DirName , Recurse Parameters DirName Type: String Name of the directory to delete, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Recurse Type: Boolean This parameter determines whether to recurse into subdirectories. If omitted, it defaults to 0 (false). Specify one of the following values: 0 (false): Do not remove files and sub-directories contained in DirName. In this case, if DirName is not empty, no action is taken and an exception is thrown. 1 (true): Remove all files and subdirectories (like the Windows command "rmdir /S"). Error Handling An exception is thrown if an error occurs. Related DirCreate, FileDelete Examples Removes the directory, but only if it is empty. DirDelete "C:\Download Temp" Removes the directory including its files and subdirectories. DirDelete "C:\Download Temp", 1 DirExist - Syntax & Usage | AutoHotkey v2 DirExist Checks for the existence of a folder and returns its attributes. AttributeString := DirExist(FilePattern) Parameters FilePattern Type: String The path, folder name, or file pattern to check. FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. Return Value Type: String This function returns the attributes of the first matching folder. This string is a subset of RASHDOC, where each letter means the following: R = READONLY A = ARCHIVE S = SYSTEM H = HIDDEN D = DIRECTORY O = OFFLINE C = COMPRESSED Since this function only checks for the existence of a folder, "D" is always present in the return value. If no folder is found, an empty string is returned. Remarks Note that searches such as DirExist ("FolderWithFilesAndSubfolders*") only tells you whether a folder exists. If you want to check for the existence of files and folders, use FileExist instead. Unlike FileGetAttrib, DirExist supports wildcard patterns and always returns a non-empty value if a matching folder exists. Since an empty string is seen as "false", the function's return value can always be used as a quasi-boolean value. For example, the statement if DirExist("C:\MyFolder") would be true if the folder exists and false otherwise. Since FilePattern may contain wildcards, DirExist may be unsuitable for validating a folder path which is to be used with another function or program. For example, DirExist("Program*") may return attributes even though "Program*" is not a valid folder name. In such cases, FileGetAttrib is preferred. Related FileExist, FileGetAttrib, File-loops Examples Shows a message box if a folder does exist. if DirExist("C:\Windows") MsgBox "The target folder does exist." Shows a message box if at least one program folder does exist. if DirExist("C:\Program*") MsgBox "At least one program folder exists." Shows a message box if a folder does not exist. if not DirExist("C:\Temp") MsgBox "The target folder does not exist." Demonstrates how to check a folder for a specific attribute. if InStr (DirExist("C:\System Volume Information"), "H") MsgBox "The folder is hidden." DirMove - Syntax & Usage | AutoHotkey v2 DirMove Moves a folder along with all its sub-folders and files. It can also rename a folder. DirMove Source, Dest , Flag Parameters Source Type: String Name of the source directory (with no trailing backslash), which is assumed to be in A_WorkingDir if an absolute path isn't specified. For example: C:\My Folder Dest Type: String The new path and name of the directory (with no trailing baskslash), which is assumed to be in A_WorkingDir if an absolute path isn't specified. For example: D:\My Folder. Note: Dest is the actual path and name that the directory will have after it is moved; it is not the directory into which Source is moved (except for the known limitation mentioned below). Flag Type: String Specify one of the following single characters: 0 (default): Do not overwrite existing files. The operation will fail if Dest already exists as a file or directory. 1: Overwrite existing files. However, any files or subfolders inside Dest that do not have a counterpart in Source will not be deleted. Known limitation: If Dest already exists as a folder and it is on the same volume as Source, Source will be moved into it rather than overwriting it. To avoid this, see the next option. 2: The same as mode 1 above except that the limitation is absent. R: Rename the directory rather than moving it. Although renaming normally has the same effect as moving, it is helpful in cases where you want "all or none" behavior; that is, when you don't want the operation to be only partially successful when Source or one of its files is locked (in use). Although this method cannot move Source onto a different volume, it can move it to any other directory on its own volume. The operation will fail if Dest already exists as a file or directory. Error Handling An exception is thrown if an error occurs. Remarks DirMove moves a single folder to a new location. To instead move the contents of a folder (all its files and subfolders), see the examples section of FileMove. If the source and destination are on different volumes or UNC paths, a copy/delete operation will be performed rather than a move. Related DirCopy, FileCopy, FileMove, FileDelete, File-loops, DirSelect, SplitPath Examples Moves a directory to a new drive. DirMove "C:\My Folder", "D:\My Folder" Performs a simple rename. DirMove "C:\My Folder", "C:\My Folder (renamed)", "R" Directories can be "renamed into" another location as long as it's on the same volume. DirMove "C:\My Folder", "C:\New Location\My Folder", "R" DirSelect - Syntax & Usage | AutoHotkey v2 DirSelect Displays a standard dialog that allows the user to select a folder. SelectedFolder := DirSelect(StartingFolder, Options, Prompt) Parameters StartingFolder Type: String If blank or omitted, the dialog's initial selection will be the user's My Documents folder (or possibly My Computer). A CLSID folder such as "::{20d04fe0-3aea-1069-a2d8-08002b30309d}" (i.e. My Computer) may be specified start navigation at a specific special folder. Otherwise, the most common usage of this parameter is an asterisk followed immediately by the absolute path of the drive or folder to be initially selected. For example, "*C:\" would initially select the C drive. Similarly, "*C:\My Folder" would initially select that particular folder. The asterisk indicates that the user is permitted to navigate upward (closer to the root) from the starting folder. Without the asterisk, the user would be forced to select a folder inside StartingFolder (or StartingFolder itself). One benefit of omitting the asterisk is that StartingFolder is initially shown in a tree-expanded state, which may save the user from having to click the first plus sign. If the asterisk is present, upward navigation may optionally be restricted to a folder other than Desktop. This is done by preceding the asterisk with the absolute path of the uppermost folder followed by exactly one space or tab. For example, "C:\My Folder *C:\My Folder\Projects" would not allow the user to navigate any higher than C:\My Folder (but the initial selection would be C:\My Folder\Projects). Options Type: Integer One of the following numbers: 0: The options below are all disabled (except on Windows 2000, where the "make new folder" button might appear anyway). 1 (default): A button is provided that allows the user to create new folders. Add 2 to the above number to provide an edit field that allows the user to type the name of a folder. For example, a value of 3 for this parameter provides both an edit field and a "make new folder" button. Add 4 to the above number to omit the BIF_NEWDIALOGSTYLE property. Adding 4 ensures that DirSelect will work properly even in a Preinstallation Environment like WinPE or BartPE. However, this prevents the appearance of a "make new folder" button. If the user types an invalid folder name in the edit field, SelectedFolder will be set to the folder selected in the navigation tree rather than what the user entered. Prompt Type: String Text displayed in the window to instruct the user what to do. If omitted or blank, it will default to "Select Folder - " A_ScriptName (i.e. the name of the current script). Return Value Type: String This function returns the full path and name of the folder chosen by the user. If the user cancels the dialog (i.e. does not wish to select a folder), an empty string is returned. If the user selects a root directory (such as C:\), the return value will contain a trailing backslash. If this is undesirable, remove it as follows: Folder := RegExReplace(DirSelect(), "\\$") ; Removes the trailing backslash, if present. An empty string is also returned if the system refused to show the dialog, but this is very rare. Remarks A folder-selection dialog usually looks like this: A GUI window may display a modal folder-selection dialog by means of the +OwnDialogs option. A modal dialog prevents the user from interacting with the GUI window until the dialog is dismissed. Known limitation: A timer that launches during the display of a DirSelect dialog will postpone the effect of the user's clicks inside the dialog until after the timer finishes. To work around this, avoid using timers whose subroutines take a long time to finish, or disable all timers during the dialog: Thread "NoTimers" SelectedFolder := DirSelect(, 3) Thread "NoTimers", false Related FileSelect, MsgBox, InputBox, ToolTip, GUI, CLSID List, DirCopy, DirMove, SplitPath Also, the operating system offers standard dialog boxes that prompt the user to pick a font, color, or icon. These dialogs can be displayed via DllCall as demonstrated at GitHub. Examples Allows the user to select a folder and provides both an edit field and a "make new folder" button. SelectedFolder := DirSelect(, 3) if SelectedFolder = "" MsgBox "You didn't select a folder." else MsgBox "You selected folder '" SelectedFolder "'." A CLSID example. Allows the user to select a folder in the "My Computer" directory. SelectedFolder := DirSelect("::{20d04fe0-3aea-1069-a2d8-08002b30309d}") ; My Computer. DllCall - Syntax & Usage | AutoHotkey v2 DllCall Calls a function inside a DLL, such as a standard Windows API function. Result := DllCall("DllFile\Function" , Type1, Arg1, Type2, Arg2, "Cdecl ReturnType") Parameters [DllFile\]Function Type: String or Integer The DLL or EXE file name followed by a backslash and the name of the function. For example: "MyDLL\MyFunction" (the file extension ".dll" is the default when omitted). If an absolute path isn't specified, DllFile is assumed to be in the system's PATH or A_WorkingDir. Note that DllCall expects a path with

backslashes (\). Forward slashes (/) are not supported. DllFile may be omitted when calling a function that resides in User32.dll, Kernel32.dll, ComCtl32.dll, or Gdi32.dll. For example, "User32\IsWindowVisible" produces the same result as "IsWindowVisible". If no function can be found by the given name, a "W" (Unicode) suffix is automatically appended. For example, "MessageBox" is the same as "MessageBoxW". Performance can be dramatically improved when making repeated calls to a DLL by loading it beforehand. This parameter may also consist solely of an integer, which is interpreted as the address of the function to call. Sources of such addresses include COM and CallbackCreate. If this parameter is an object, the value of the object's Ptr property is used. If no such property exists, a PropertyError is thrown. Type1, Arg1 Type: String Each of these pairs represents a single parameter to be passed to the function. The number of pairs is unlimited. For Type, see the types table below. For Arg, specify the value to be passed to the function. Cdecl ReturnType Type: String The word Cdecl is normally omitted because most functions use the standard calling convention rather than the "C" calling convention (functions such as wsprintf that accept a varying number of arguments are one exception to this). Note that most object-oriented C++ functions use the thiscall convention, which is not supported. If present, the word Cdecl should be listed before the return type (if any). Separate each word from the next with a space or tab. For example: "Cdecl Str". Since a separate "C" calling convention does not exist in 64-bit code, Cdecl may be specified but has no effect on 64-bit builds of AutoHotkey. ReturnType: If the function returns a 32-bit signed integer (Int), BOOL, or nothing at all, ReturnType may be omitted. Otherwise, specify one of the argument types from the types table below. The asterisk suffix is also supported. Return Value Type: String or Integer DllCall returns the actual value returned by Function. If Function is of a type that does not return a value, the result is an undefined value of the specified return type (integer by default). Types of Arguments and Return Values Type Description Str A string such as "Blue" or MyVar, or a VarRef such as &MyVar. If the called function modifies the string and the argument is a naked variable or VarRef, its contents will be updated. For example, the following call would convert the contents of MyVar to uppercase: DllCall("CharUpper", "Str", MyVar). If the function is designed to store a string longer than the parameter's input value (or if the parameter is for output only), the recommended approach is to create a Buffer, use the Ptr type to pass it, and use StrGet to retrieve the string after the function returns, as in the wsprintf example. Otherwise, ensure that the variable is large enough before calling the function. This can be achieved by calling VarSetStrCapacity(MyVar, 123), where 123 is the number of 16-bit units (loosely referred to as characters) that MyVar must be able to hold. If the variable is not null-terminated upon return, an error message is shown and the program exits as it is likely that memory has been corrupted via buffer overrun. This would typically indicate that the variable's capacity was insufficient. A Str argument must not be an expression that evaluates to a number (such as i+1). If it is, the function is not called and a TypeError is thrown. The rarely-used Str* arg type passes the address of a temporary variable containing the address of the string. If the function writes a new address into the temporary variable, the new string is copied into the script's variable, if a VarRef was passed. This can be used with functions that expect something like "TCHAR **" or "LPTSTR *". However, if the function allocates memory and expects the caller to free it (such as by calling CoTaskMemFree), the Ptr* arg type must be used instead. Note: When passing a string to a function, be aware what type of string the function expects. WStr Since AutoHotkey uses UTF-16 natively, WStr (wide character string) is equivalent to Str. AStr AStr causes the input value to be automatically converted to ANSI. Since the temporary memory used for this conversion is only large enough for the converted input string, any value written to it by the function is discarded. To receive an ANSI string as an output parameter, follow this example: buf := Buffer(length) ; Allocate temporary buffer. DllCall("Function", "ptr", buf) ; Pass buffer to function. str := StrGet(buf, "cp0") ; Retrieve ANSI string from buffer. The rarely-used AStr* arg type is also supported and behaves similarly to the Str* type, except that any new string is converted from ANSI to the native format, UTF-16. See Binary Compatibility for equivalent Win32 types and other details. Int64 A 64-bit integer, whose range is -9223372036854775808 (-0x8000000000000000) to 9223372036854775807 (0x7FFFFFFFFFFFFFFF). Int A 32-bit integer (the most common integer type), whose range is -2147483648 (-0x80000000) to 2147483647 (0x7FFFFFFF). An Int is sometimes called a "Long". An Int should also be used for each BOOL argument expected by a function (a BOOL value should be either 1 or 0). An unsigned Int (UInt) is also used quite frequently, such as for DWORD. Short A 16-bit integer, whose range is -32768 (-0x8000) to 32767 (0x7FFF). An unsigned Short (UShort) can be used with functions that expect a WORD. Char An 8-bit integer, whose range is -128 (-0x80) to 127 (0x7F). An unsigned character (UChar) can be used with functions that expect a BYTE. Float A 32-bit floating point number, which provides 6 digits of precision. Double A 64-bit floating point number, which provides 15 digits of precision. Ptr A pointer-sized integer, equivalent to Int or Int64 depending on whether the exe running the script is 32-bit or 64-bit. Ptr should be used for pointers to arrays or structures (such as RECT* or LPPOINT) and almost all handles (such as HWND, HBRUSH or HBITMAP). If the parameter is a pointer to a single numeric value such as LPDWORD or int*, generally the * or P suffix should be used instead of "Ptr". If an object is passed to a Ptr parameter, the value of the object's Ptr property is used. If no such property exists, a PropertyError is thrown. Typically the object would be a Buffer. If an object is passed to a Ptr* parameter, the value of the object's Ptr property is retrieved before the call and the address of a temporary variable containing this value is passed to the function. After the function returns, the new value is assigned back to the object's Ptr property. Ptr can also be used with the * or P suffix; it should be used with functions that output a pointer via LPVOID* or similar. UPtr is also valid, but is only unsigned in 32-bit builds as AutoHotkey does not support unsigned 64-bit integers. Note: To pass a NULL handle or pointer, pass the integer 0. * or P (suffix) Append an asterisk (with optional preceding space) to any of the above types to cause the address of the argument to be passed rather than the value itself (the called function must be designed to accept it). Since the value of such an argument might be modified by the function, whenever a VarRef is passed as the argument, the variable's contents will be updated after the function returns. For example, the following call would pass the contents of MyVar to MyFunction by address, but would also update MyVar to reflect any changes made to it by MyFunction: DllCall ("MyDll\MyFunction", "Int*", &MyVar). In general, an asterisk is used whenever a function has an argument type or return type that starts with "LP". The most common example is LPDWORD, which is a pointer to a DWORD. Since a DWORD is an unsigned 32-bit integer, use "UInt*" or "UIntP" to represent LPDWORD. An asterisk should not be used for string types such as LPTSTR, pointers to structures such as LPRECT, or arrays; for these, "Str" or "Ptr" should be used, depending on whether you pass a string, address or Buffer. Note: "Char*" is not the same as "Str" because "Char*" passes the address of an 8-bit number, whereas "Str" passes the address of a series of characters, which may be either 16-bit (Unicode) or 8-bit (for "AStr"), depending on the version of AutoHotkey. Similarly, "UInt*" passes the address of a 32-bit number, so should not be used if the function expects an array of values or a structure larger than 32 bits. Since variables in AutoHotkey have no fixed type, the address passed to the function points to temporary memory rather than the caller's variable. U (prefix) Prepend the letter U to any of the integer types above to interpret it as an unsigned integer (UInt64, UInt, UShort, and UChar). Strictly speaking, this is necessary only for return values and asterisk variables because it does not matter whether an argument passed by value is unsigned or signed (except for Int64). If a negative integer is specified for an unsigned argument, the integer wraps around into the unsigned domain. For example, when -1 is sent as a UInt, it would become 0xFFFFFFFF. Unsigned 64-bit integers produced by a function are not supported. Therefore, to work with numbers greater or equal to 0x8000000000000000, omit the U prefix and interpret any negative values received from the function as large integers. For example, a function that yields -1 as an Int64 is really yielding 0xFFFFFFFFFFFFFFFF if it is designed to yield a UInt64. HRESULT A 32-bit integer. This is generally used with COM functions and is valid only as a return type without any prefix or suffix. Error values (as defined by the FAILED macro) are never returned; instead, an OSError is thrown. Therefore, the return value is a success code in the range 0 to 2147483647. HRESULT is the default return type for ComCall. Errors DllCall throws an Error under any of the following conditions: OSError: The HRESULT return type was used and the function returned an error value (as defined by the FAILED macro). Exception.Extra contains the hexadecimal error code. TypeError: The [DllFile\|]Function parameter is a floating point number. A string or positive integer is required. ValueError: The return type or one of the specified arg types is invalid. TypeError: An argument was passed a value of an unexpected type. For instance, an expression that evaluates to a number was passed to a string (str) argument, a non-numeric string was passed to a numeric argument, or an object was passed to an argument not of type Ptr. Error: The specified DllFile could not be accessed or loaded. If no explicit path was specified for DllFile, the file must exist in the system's PATH or A_WorkingDir. This error might also occur if the user lacks permission to access the file, or if AutoHotkey is 32-bit and the DLL is 64-bit or vice versa. Error: The specified function could not be found inside the DLL. Error: The function was called but it aborted with a fatal exception. Exception.Extra contains the exception code. For example, 0xC0000005 means "access violation". In such cases, the thread is aborted (if try is not used), but any asterisk variables are still updated. An example of a fatal exception is dereferencing an invalid pointer such as NULL (0). Since a Cdecl function never produces the error described in the next paragraph, it may generate an exception when too few arguments are passed to it. Error: The function was called but was passed too many or too few arguments. Exception.Extra contains the number of bytes by which the argument list was incorrect. If it is positive, too many arguments (or arguments that were too large) were passed, or the call requires CDecl. If it is negative, too few arguments were passed. This situation should be corrected to ensure reliable operation of the function. The presence of this error may also indicate that an exception occurred. Note that due to the x64 calling convention, 64-bit builds never raise this error. Native Exceptions and A_LastError In spite of the built-in exception handling, it is still possible to crash a script with DllCall. This can happen when a function does not directly generate an exception but yields something inappropriate, such as a bad pointer or a string that is not terminated. This might not be the function's fault if the script passed it an unsuitable value such as a bad pointer or a "str" with insufficient capacity. A script can also crash when it specifies an inappropriate argument type or return type, such as claiming that an ordinary integer yielded by a function is an asterisk variable or str. The built-in variable A_LastError contains the result of the operating system's GetLastError() function. Performance When making repeated calls to a DLL, performance can be dramatically improved by loading it explicitly (this is not necessary for a standard DLL such as User32 because it is always resident). This practice avoids the need for DllCall to internally call LoadLibrary and FreeLibrary each time. For example: hModule := DllCall("LoadLibrary", "Str", "MyFunctions.dll", "Ptr") ; Avoids the need for DllCall in the loop to load the library. Loop Files, "C:\My Documents*.*", "R" result := DllCall("MyFunctions\BackupFile", "Str", A_LoopFilePath) DllCall("FreeLibrary", "Ptr", hModule) ; To conserve memory, the DLL may be unloaded after using it. Even faster performance can be achieved by looking up the function's address beforehand. For example: ; In the following example, if the DLL isn't yet loaded, use LoadLibrary in place of GetModuleHandle. MulDivProc := DllCall("GetProcAddress", "Ptr",

DllCall("GetModuleHandle", "Str", "kernel32", "Ptr"), "AStr", "MulDiv", "Ptr") Loop 500 DllCall(MulDivProc, "Int", 3, "Int", 4, "Int", 3) If DllCall's first parameter is a literal string such as "MulDiv" and the DLL containing the function is ordinarily loaded before the script starts, or has been successfully loaded with #DllLoad, the string is automatically resolved to a function address. This built-in optimization is more effective than the example shown above. Finally, when passing a string-variable to a function that will not change the length of the string, performance is improved by passing the variable by address (e.g. StrPtr(MyVar)) rather than as a "str" (especially when the string is very long). The following example converts a string to uppercase: DllCall("CharUpper", "Ptr", StrPtr(MyVar), "Ptr"). Structures and Arrays A structure is a collection of members (fields) stored adjacently in memory. Most members tend to be integers. Functions that accept the address of a structure (or a memory-block array) can be called by allocating memory by some means and passing the memory address to the function. The Buffer object is recommended for this purpose. The following steps are generally used: 1) Call MyStruct := Buffer(123, 0) to allocate a buffer to hold the structure's data. Replace 123 with a number that is at least as large as the size of the structure, in bytes. Specifying zero as the last parameter is optional; it initializes all members to be binary zero, which is typically used to avoid calling NumPut as often in the next step. 2) If the target function uses the values initially in the structure, call NumPut("UInt", 123, MyStruct, 4) to initialize any members that should be non-zero. Replace 123 with the integer to be put into the target member (or specify StrPtr(Var) to store the address of a string). Replace 4 with the offset of the target member (see step #4 for description of "offset"). Replace "UInt" with the appropriate type, such as "Ptr" if the member is a pointer or handle. 3) Call the target function, passing MyStruct as a Ptr argument. For example, DllCall ("MyDll\MyFunc", "Ptr", MyStruct). The function will examine and/or change some of the members. DllCall automatically uses the address of the buffer, which is normally retrieved by using MyStruct.Ptr. 4) Use MyInteger := NumGet(MyStruct, 4, "UInt") to retrieve any desired integers from the structure. Replace 4 with the offset of the target member in the structure. The first member is always at offset 0. The second member is at offset 0 plus the size of the first member (typically 4). Members beyond the second are at the offset of the previous member plus the size of the previous member. Most members -- such as DWORD, Int, and other types of 32-bit integers -- are 4 bytes in size. Replace "UInt" with the appropriate type or omit it if the member is a pointer or handle. See Structure Examples for actual usages. Known Limitations When a variable's string address (e.g. StrPtr(MyVar)) is passed to a function and that function alters the length of the variable's contents, subsequent uses of the variable may behave incorrectly. To fix this, do one of the following: 1) Pass MyVar as a "Str" argument rather than as a Ptr/address; 2) Call VarSetStrCapacity(MyVar, -1) to update the variable's internally-stored length after calling DllCall. Any binary zero stored in a variable by a function may act as a terminator, preventing all data to the right of the zero from being accessed or changed by most built-in functions. However, such data can be manipulated by retrieving the string's address with StrPtr and passing it to other functions, such as NumPut, NumGet, StrGet, StrPut, and DllCall itself. A function that returns the address of one of the strings that was passed into it might return an identical string in a different memory address than expected. For example calling CharLower(CharUpper(MyVar)) in a programming language would convert MyVar's contents to lowercase. But when the same is done with DllCall, MyVar would be uppercase after the following call because CharLower would have operated on a different/temporary string whose contents were identical to MyVar: MyVar := "ABC" result := DllCall("CharLower", "Str", DllCall("CharUpper", "Str", MyVar, "Str"), "Str") To work around this, change the two underlined "Str" values above to Ptr. This interprets CharUpper's return value as a pure address that will get passed as an integer to CharLower. Certain limitations may be encountered when dealing with strings. For details, see Binary Compatibility. Component Object Model (COM) COM objects which are accessible to VBScript and similar languages are typically also accessible to AutoHotkey via ComObject, ComObjGet or ComObjActive and the built-in object syntax. COM objects which don't support IDispatch can be used with DllCall by retrieving the address of a function from the virtual function table of the object's interface. For more details, see the example further below. However, it is usually better to use ComCall, which streamlines this process. .NET Framework .NET Framework libraries are executed by a "virtual machine" known as the Common Language Runtime, or CLR. That being the case, .NET DLL files are formatted differently to normal DLL files, and generally do not contain any functions which DllCall is capable of calling. However, AutoHotkey can utilize the CLR through COM callable wrappers. Unless the library is also registered as a general COM component, the CLR itself must first be manually initialized via DllCall. For details, see .NET Framework Interop (which currently utilizes DllCall and is not compatible with AutoHotkey v2). Related Binary Compatibility, Buffer object, ComCall, PostMessage, OnMessage, CallbackCreate, Run, VarSetStrCapacity, Functions, SysGet, #DllLoad, Windows API Index Examples Calls the Windows API function "MessageBox" and reports which button the user presses. WhichButton := DllCall("MessageBox", "Int", 0, "Str", "Press Yes or No", "Str", "Title of box", "Int", 4) MsgBox "You pressed button #" WhichButton Changes the desktop wallpaper to the specified bitmap (.bmp) file. DllCall("SystemParametersInfo", "UInt", 0x14, "UInt", 0, "Str", A_WinDir . "\winnt.bmp", "UInt", 1) Calls the API function "IsWindowVisible" to find out if a Notepad window is visible. DetectHiddenWindows True if not DllCall("IsWindowVisible", "Ptr", WinExist("Untitled - Notepad")) ; WinExist returns an HWND. MsgBox "The window is not visible." Calls the API's wsprintf() to pad the number 432 with leading zeros to make it 10 characters wide (0000000432). ZeroPaddedNumber := Buffer(20) ; Ensure the buffer is large enough to accept the new string. DllCall("wsprintf", "Ptr", ZeroPaddedNumber, "Str", "%010d", "Int", 432, "Cdecl") ; Requires the Cdecl calling convention. MsgBox StrGet(ZeroPaddedNumber) ; Alternatively, use the Format function in conjunction with the zero flag: MsgBox Format ("{:010}", 432) Demonstrates QueryPerformanceCounter(), which gives more precision than A_TickCount's 10ms. DllCall("QueryPerformanceFrequency", "Int64*", &freq := 0) DllCall("QueryPerformanceCounter", "Int64*", &CounterBefore := 0) Sleep 1000 DllCall("QueryPerformanceCounter", "Int64*", &CounterAfter := 0) MsgBox "Elapsed QPC time is " . (CounterAfter - CounterBefore) / freq * 1000 " ms" Press a hotkey to temporarily reduce the mouse cursor's speed, which facilitates precise positioning. Hold down F1 to slow down the cursor. Release it to return to original speed. F1:: F1 up:: { static SPI_GETMOUSESPEED := 0x70 static SPI_SETMOUSESPEED := 0x71 static OrigMouseSpeed := 0 switch ThisHotkey { case "F1": ; Retrieve the current speed so that it can be restored later: DllCall("SystemParametersInfo", "UInt", SPI_GETMOUSESPEED, "UInt", 0, "Ptr*", OrigMouseSpeed, "UInt", 0) ; Now set the mouse to the slower speed specified in the next-to-last parameter (the range is 1-20, 10 is default): DllCall("SystemParametersInfo", "UInt", SPI_SETMOUSESPEED, "UInt", 0, "Ptr", 3, "UInt", 0) KeyWait "F1" ; This prevents keyboard auto-repeat from doing the DllCall repeatedly. case "F1 up": DllCall("SystemParametersInfo", "UInt", SPI_SETMOUSESPEED, "UInt", 0, "Ptr", OrigMouseSpeed, "UInt", 0) ; Restore the original speed. } } Monitors the active window and displays the position of its vertical scroll bar in its focused control (with real-time updates). SetTimer WatchScrollBar, 100 WatchScrollBar() { FocusedHwnd := 0 try FocusedHwnd := ControlGetFocus("A") if !FocusedHwnd ; No focused control. return ; Display the vertical or horizontal scroll bar's position in a tooltip: ToolTip DllCall("GetScrollPos", "Ptr", FocusedHwnd, "Int", 1) ; Last parameter is 1 for SB_VERT, 0 for SB_HORZ. } Writes some text to a file then reads it back into memory. This method can be used to help performance in cases where multiple files are being read or written simultaneously. Alternatively, FileOpen can be used to achieve the same effect. FileName := FileSelect("S16",, "Create a new file:") if FileName = "" return GENERIC_WRITE := 0x40000000 ; Open the file for writing rather than reading. CREATE_ALWAYS := 2 ; Create new file (overwriting any existing file). hFile := DllCall("CreateFile", "Str", FileName, "UInt", GENERIC_WRITE, "UInt", 0, "Ptr", 0, "UInt", CREATE_ALWAYS, "UInt", 0, "Ptr", 0, "Ptr") if !hFile { MsgBox "Can't open '" FileName "' for writing." return } TestString := "This is a test string.`r`n" ; When writing a file this way, use `r`n rather than `n to start a new line. StrSize := StrLen(TestString) * 2 DllCall("WriteFile", "Ptr", hFile, "Str", TestString, "UInt", StrSize, "UIntP", &BytesActuallyWritten := 0, "Ptr", 0) DllCall ("CloseHandle", "Ptr", hFile) ; Close the file. ; Now that the file was written, read its contents back into memory. GENERIC_READ := 0x80000000 ; Open the file for reading rather than writing. OPEN_EXISTING := 3 ; This mode indicates that the file to be opened must already exist. FILE_SHARE_READ := 0x1 ; This and the next are whether other processes can open the file while we have it open. FILE_SHARE_WRITE := 0x2 hFile := DllCall("CreateFile", "Str", FileName, "UInt", GENERIC_READ, "UInt", FILE_SHARE_READ|FILE_SHARE_WRITE, "Ptr", 0, "UInt", OPEN_EXISTING, "UInt", 0, "Ptr", 0) if !hFile { MsgBox "Can't open '" FileName "' for reading." return } ; Allocate a block of memory for the string to read: Buf := Buffer(StrSize) DllCall("ReadFile", "Ptr", hFile, "Ptr", Buf, "UInt", Buf.Size, "UIntP", &BytesActuallyRead := 0, "Ptr", 0) DllCall("CloseHandle", "Ptr", hFile) ; Close the file. MsgBox "The following string was read from the file: " StrGet(Buf) Hides the mouse cursor when you press Win+C. To later show the cursor, press this hotkey again. OnExit (*) => SystemCursor ("Show") ; Ensure the cursor is made visible when the script exits. #c::SystemCursor("Toggle") ; Win+C hotkey to toggle the cursor on and off. SystemCursor (cmd) ; cmd = "Show|Hide|Toggle|Reload" { static visible := true, c := Map() static sys_cursors := [32512, 32513, 32514, 32515, 32516, 32642 , 32643, 32644, 32645, 32646, 32648, 32649, 32650] if (cmd = "Reload" or !c.Count) ; Reload when requested or at first call. { for i, id in sys_cursors { h_cursor := DllCall("LoadCursor", "Ptr", 0, "Ptr", id) h_default := DllCall("CopyImage", "Ptr", h_cursor, "UInt", 2 , "Int", 0, "Int", 0, "UInt", 0) h_blank := DllCall("CreateCursor", "Ptr", 0, "Int", 0, "Int", 0 , "Int", 32, "Int", 32 , "Ptr", Buffer(32*4, 0xFF) , "Ptr", Buffer(32*4, 0)) c[id] := {default: h_default, blank: h_blank} } } switch cmd { case "Show": visible := true case "Hide": visible := false case "Toggle": visible := !visible default: return } for id, handles in c { h_cursor := DllCall("CopyImage" , "Ptr", visible ? handles.default : handles.blank , "UInt", 2, "Int", 0, "Int", 0, "UInt", 0) DllCall("SetSystemCursor", "Ptr", h_cursor, "UInt", id) } } Structure example. Pass the address of a RECT structure to GetWindowRect(), which sets the structure's members to the positions of the left, top, right, and bottom sides of a window (relative to the screen). Run "Notepad" WinWait "Untitled - Notepad" ; This also sets the "last found window" for use with WinExist below. Rect := Buffer(16) ; A RECT is a struct consisting of four 32-bit integers (i.e. 4*4=16). DllCall("GetWindowRect", "Ptr", WinExist(), "Ptr", Rect) ; WinExist returns an HWND. L := NumGet(Rect, 0, "Int"), T := NumGet(Rect, 4, "Int") R := NumGet(Rect, 8, "Int"), B := NumGet(Rect, 12, "Int") MsgBox Format("Left {1} Top {2} Right {3} Bottom {4}", L, T, R, B) Structure example. Pass to FillRect() the address of a RECT structure that indicates a part of the screen to temporarily paint red. Rect := Buffer(16) ; Set capacity to hold four 4-byte integers. NumPut("Int", 0 ; left , "Int", 0 ; top , "Int", A_ScreenWidth//2 ; right , "Int", A_ScreenHeight//2 ; bottom , Rect) hDC := DllCall("GetDC", "Ptr", 0, "Ptr") ; Pass zero to get the desktop's device context. hBrush := DllCall("CreateSolidBrush", "UInt", 0x0000FF, "Ptr") ; Create a red brush (0x0000FF is in BGR format). DllCall("FillRect", "Ptr", hDC, "Ptr", Rect, "Ptr", hBrush) ; Fill the specified rectangle using the brush above. DllCall("ReleaseDC", "Ptr", 0, "Ptr", hDC) ; Clean-up. DllCall("DeleteObject", "Ptr", hBrush) ; Clean-up. Structure example. Changes the system's clock

to the specified date and time. Use caution when changing to a date in the future as it may cause scheduled tasks to run prematurely! SetSystemTime ("20051008142211") ; Pass it a timestamp (local, not UTC). SetSystemTime(YYYYMMDDHHMISS) ; Sets the system clock to the specified date and time. ; Caller must ensure that the incoming parameter is a valid date-time stamp ; (local time, not UTC). Returns non-zero upon success and zero otherwise. { ; Convert the parameter from local time to UTC for use with SetSystemTime(). UTC_Delta := DateDiff(A_Now, A_NowUTC, "Seconds") ; Seconds is more accurate due to rounding issue. UTC_Delta := Round(-UTC_Delta/60) ; Round to nearest minute to ensure accuracy. YYYYMMDDHHMISS := DateAdd(YYYYMMDDHHMISS, UTC_Delta, "Minutes") ; Apply offset to convert to UTC. SystemTime := Buffer(16) ; This struct consists of 8 UShorts (i.e. 8*2=16). NumPut("UShort", SubStr (YYYYMMDDHHMISS, 1, 4) ; YYYY (year) , "UShort", SubStr(YYYYMMDDHHMISS, 5, 2) ; MM (month of year, 1-12) , "UShort", 0 ; Unused (day of week) , "UShort", SubStr(YYYYMMDDHHMISS, 7, 2) ; DD (day of month) , "UShort", SubStr(YYYYMMDDHHMISS, 9, 2) ; HH (hour in 24-hour time) , "UShort", SubStr(YYYYMMDDHHMISS, 11, 2) ; MI (minute) , "UShort", SubStr(YYYYMMDDHHMISS, 13, 2) ; SS (second) , "UShort", 0 ; Unused (millisecond) , SystemTime) return DllCall("SetSystemTime", "Ptr", SystemTime) } More structure examples: See the WinLIRC client script for a demonstration of how to use DllCall to make a network connection to a TCP/IP server and receive data from it. The operating system offers standard dialog boxes that prompt the user to pick a font and/or color, or an icon. These dialogs use structures and are demonstrated at GitHub. Removes the active window from the taskbar for 3 seconds. Compare this to the equivalent ComCall example. /* Methods in ITaskbarList's VTable: IUnknown: 0 QueryInterface -- use ComObjQuery instead 1 AddRef -- use ObjAddRef instead 2 Release -- use ObjRelease instead ITaskbarList: 3 HrInit 4 AddTab 5 DeleteTab 6 ActivateTab 7 SetActiveAlt */ IID_ITaskbarList := "{56FDF342-FD6D-11d0-958A-006097C9A090}" CLSID_TaskbarList := "{56FDF344-FD6D-11d0-958A-006097C9A090}" ; Create the TaskbarList object. tbl := ComObject(CLSID_TaskbarList, IID_ITaskbarList) activeHwnd := WinExist("A") DllCall(vtable(tbl.ptr,3), "ptr", tbl) ; tbl.HrInit() DllCall(vtable(tbl.ptr,5), "ptr", tbl, "ptr", activeHwnd) ; tbl.DeleteTab(activeHwnd) Sleep 3000 DllCall(vtable(tbl.ptr,4), "ptr", tbl, "ptr", activeHwnd) ; tbl.AddTab(activeHwnd) ; Non-wrapped interface pointers must be manually freed. ObjRelease(tbl.ptr) vtable(ptr, n) { ; NumGet(ptr, "ptr") returns the address of the object's virtual function ; table (vtable for short). The remainder of the expression retrieves ; the address of the nth function's address from the vtable. return NumGet(NumGet(ptr, "ptr"), n*A_PtrSize, "ptr") } Download - Syntax & Usage | AutoHotkey v2 Download Downloads a file from the Internet. Download URL, Filename Parameters URL Type: String URL of the file to download. For example, http://someorg.org might retrieve the welcome page for that organization. Filename Type: String Specify the name of the file to be created locally, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Any existing file will be overwritten by the new file. Error Handling An exception is thrown on failure. Remarks This function downloads to a file. To download to a variable instead, see the example below. For alternative methods, see the following forum topic: www.autohotkey.com/forum/topic10466.html The download might appear to succeed even when the remote file doesn't exist. This is because many web servers send an error page instead of the missing file. This error page is what will be saved in place of Filename. Firewalls or the presence of multiple network adapters may cause this function to fail. Also, some websites may block such downloads. Caching: By default, the URL is retrieved directly from the remote server (that is, never from Internet Explorer's cache). To permit caching, precede the URL with *0 followed by a space; for example: *0 http://someorg.org. The zero following the asterisk may be replaced by any valid dwFlags number; for details, search www.microsoft.com for InternetOpenUrl. Proxies: If a proxy server has been configured in Microsoft Internet Explorer's settings, it will be used. FTP and Gopher URLS are also supported. For example: Download "ftp://example.com/home/My File.zip", "C:\My Folder\My File.zip" ; Log in anonymously. Download "ftp://user:pass@example.com:21/home/My File.zip", "C:\My Folder\My File.zip" ; Log in as a specific user. Download "ftp://user:pass@example.com/My Directory", "C:\Dir Listing.html" ; Gets a directory listing in HTML format. Related FileRead, FileCopy Examples Downloads a text file. Download "https://www.autohotkey.com/download/2.0/version.txt", "C:\AutoHotkey Latest Version.txt" Downloads a zip file. Download "http://someorg.org/archive.zip", "C:\SomeOrg's Archive.zip" Downloads text to a variable. whr := ComObject("WinHttp.WinHttpRequest.5.1") whr.Open("GET", "https://www.autohotkey.com/download/2.0/version.txt", true) whr.Send() ; Using 'true' above and the call below allows the script to remain responsive. whr.WaitForResponse() version := whr.ResponseText MsgBox version Makes an asynchronous HTTP request. req := ComObject("Msxml2.XMLHTTP") ; Open a request with async enabled. req.open("GET", "https://www.autohotkey.com/download/2.0/version.txt", true) ; Set our callback function. req.onreadystatechange := Ready ; Send the request. Ready() will be called when it's complete. req.send() /* ; If you're going to wait, there's no need for onreadystatechange. ; Setting async=true and waiting like this allows the script to remain ; responsive while the download is taking place, whereas async=false ; will make the script unresponsive. while req.readyState != 4 sleep 100 */ Persistent Ready() { if (req.readyState != 4) ; Not done yet. return if (req.status == 200) ; OK. MsgBox "Latest AutoHotkey version: " req.responseText else MsgBox "Status " req.status,, 16 ExitApp } List of Drive Functions | AutoHotkey v2 Drive Functions Functions for retrieving various types of information about the computer's drive (s), or making a variety of changes to a drive. Click on a function name for details. Function Description DriveEject Ejects the tray of the specified CD/DVD drive, or ejects a removable drive. DriveGetCapacity Returns the total capacity of the drive which contains the specified path, in megabytes. DriveGetFileSystem Returns the type of the specified drive's file system. DriveGetLabel Returns the volume label of the specified drive. DriveGetList Returns a string of letters, one character for each drive letter in the system. DriveGetSerial Returns the volume serial number of the specified drive. DriveGetSpaceFree Returns the free disk space of the drive which contains the specified path, in megabytes. DriveGetStatus Returns the status of the drive which contains the specified path. DriveGetStatusCD Returns the media status of the specified CD/DVD drive. DriveGetType Returns the type of the drive which contains the specified path. DriveLock Prevents the eject feature of the specified drive from working. DriveRetract Retracts the tray of the specified CD/DVD drive. DriveSetLabel Changes the volume label of the specified drive. DriveUnlock Restores the eject feature of the specified drive. Error Handling An exception is thrown on failure. Related List of all functions Examples Allows the user to select a drive in order to analyze it. folder := DirSelect(, 3, "Pick a drive to analyze:") if not folder return MsgBox ("All Drives: " DriveGetList() " Selected Drive: " folder " Drive Type: " DriveGetType(folder) " Status: " DriveGetStatus(folder) " Capacity: " DriveGetCapacity(folder) " MB Free Space: " DriveGetSpaceFree(folder) " MB Filesystem: " DriveGetFilesystem(folder) " Volume Label: " DriveGetLabel(folder) " Serial Number: " DriveGetSerial(folder)) DriveEject / DriveRetract - Syntax & Usage | AutoHotkey v2 DriveEject / DriveRetract Ejects or retracts the tray of the specified CD/DVD drive, or ejects a removable drive. DriveEject Drive DriveRetract Drive Parameters Drive Type: String The drive letter optionally followed by a colon or a colon and backslash. For example, D, D: or D:\. This can also be a device path in the form \\?\Volume{...}, which can be discovered by running mountvol at the command line. In this case the drive is not required to be assigned a drive letter. If omitted, it defaults to the first CD/DVD drive found by iterating from A to Z. An exception is thrown if no drive is found. Error Handling An exception is thrown on failure, if detected. These functions will probably not work on a network drive or any drive lacking the "Eject" option in Explorer. The underlying system functions do not always report failure, so an exception may or may not be thrown. Remarks This function waits for the ejection or retraction to complete before allowing the script to continue. It may be possible to detect the previous tray state by measuring the time the function takes to complete, as in the example below. Ejecting a removable drive is generally equivalent to using the "Eject" context menu option in Explorer, except that no warning is shown if files are in use. Unlike the Safely Remove Hardware option, this only dismounts the volume identified by the Drive parameter, not the overall device. DriveEject and DriveRetract correspond to the IOCTL_STORAGE_EJECT_MEDIA and IOCTL_STORAGE_LOAD_MEDIA control codes, which may also have an effect on drive types other than CD/DVD, such as tape drives. Related DriveGetStatusCD, Drive functions Examples Ejects (opens) the tray of the first CD/DVD drive. DriveEject() Retracts (closes) the tray of the first CD/DVD drive. DriveRetract() Ejects all removable drives (except CD/DVD drives). Loop Parse DriveGetList("REMOVABLE") { if MsgBox("Eject " A_LoopField ":, even if files are open?",, "y/n") = "yes" DriveEject(A_LoopField) } else MsgBox "No removable drives found." Defines a hotkey which toggles the tray to the opposite state (open or closed), based on the time the function takes to complete. #c:: { DriveEject ; If the function completed quickly, the tray was probably already ejected. ; In that case, retract it: if (A_TimeSinceThisHotkey < 1000) ; Adjust this time if needed. DriveRetract } DriveGetCapacity - Syntax & Usage | AutoHotkey v2 DriveGetCapacity Returns the total capacity of the drive which contains the specified path, in megabytes. Capacity := DriveGetCapacity(Path) Parameters Path Type: String Any path contained by the drive (might also work on UNC paths and mapped drives). Return Value Type: Integer This function returns the total capacity of the drive which contains the specified path, in megabytes. Error Handling An exception is thrown on failure. Remarks In general, Path can be any path. Since NTFS supports mounted volumes and directory junctions, different paths with the same drive letter can produce different amounts of capacity. Related DriveGetSpaceFree, Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetFileSystem - Syntax & Usage | AutoHotkey v2 DriveGetFileSystem Returns the type of the specified drive's file system. FileSystem := DriveGetFileSystem(Drive) Parameters Drive Type: String The drive letter followed by a colon and an optional backslash, or a UNC name such \server1\share1. Return Value Type: String This function returns the type of Drive's file system. The possible values are defined by the system; they include (but are not limited to) the following: NTFS, FAT32, FAT, CDFS (typically indicates a CD), or UDF (typically indicates a DVD). Error Handling An exception is thrown on failure, such as if the drive does not contain formatted media. Related Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetLabel - Syntax & Usage | AutoHotkey v2 DriveGetLabel Returns the volume label of the specified drive. Label := DriveGetLabel(Drive) Parameters Drive Type: String The drive letter followed by a colon and an optional backslash, or a UNC name such \server1\share1. Return Value Type: String This function returns the volume label of the specified drive. Error Handling An exception is thrown on failure. Related DriveSetLabel, Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetList - Syntax & Usage | AutoHotkey v2 DriveGetList Returns a string of letters, one character for each drive letter in the system. List := DriveGetList(Type) Parameters Type Type: String If omitted, all drive types are retrieved. Otherwise, specify one of the following words to retrieve only a specific type of drive: CDROM, REMOVABLE, FIXED, NETWORK, RAMDISK, UNKNOWN. Return Value Type: String This function returns the drive letters in the system, depending on the specified type. For example: ACDEZ. Related Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetSerial - Syntax & Usage | AutoHotkey v2

DriveGetSerial Returns the volume serial number of the specified drive. Serial := DriveGetSerial(Drive) Parameters Drive Type: String The drive letter followed by a colon and an optional backslash, or a UNC name such \server1\share1. Return Value Type: Integer This function returns the volume serial number of the specified drive. See Format for how to convert it to hexadecimal. Error Handling An exception is thrown on failure. Related Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetSpaceFree - Syntax & Usage | AutoHotkey v2 DriveGetSpaceFree Returns the free disk space of the drive which contains the specified path, in megabytes. FreeSpace := DriveGetSpaceFree(Path) Parameters Path Type: String Any path contained by the drive (might also work on UNC paths and mapped drives). Return Value Type: Integer This function returns the free disk space of the drive which contains the specified path, in megabytes (rounded down to the nearest megabyte). Error Handling An exception is thrown on failure. Remarks In general, Path can be any path. Since NTFS supports mounted volumes and directory junctions, different paths with the same drive letter can produce different amounts of free space. Related DriveGetCapacity, Drive functions Examples Retrieves and reports the free disk space of the drive which contains A_MyDocuments. FreeSpace := DriveGetSpaceFree(A_MyDocuments) MsgBox FreeSpace " MB" See example #1 on the Drive Functions page for another demonstration of this function. DriveGetStatus - Syntax & Usage | AutoHotkey v2 DriveGetStatus Returns the status of the drive which contains the specified path. Status := DriveGetStatus(Path) Parameters Path Type: String Any path contained by the drive (might also work on UNC paths and mapped drives). Return Value Type: String This function returns the status of the drive which contains the specified path: Status Notes Unknown Might indicate unformatted/RAW file system. Ready This is the most common. NotReady Typical for removable drives that don't contain media. Invalid Path does not exist or is a network drive that is presently inaccessible, etc. Error Handling An exception is thrown on failure. Remarks In general, Path can be any path. Since NTFS supports mounted volumes and directory junctions, different paths with the same drive letter can produce different results. Related Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetStatusCD - Syntax & Usage | AutoHotkey v2 DriveGetStatusCD Returns the media status of the specified CD/DVD drive. CDStatus := DriveGetStatusCD(Drive) Parameters Drive Type: String The drive letter followed by a colon. If omitted, the default CD/DVD drive will be used. Return Value Type: String This function returns the media status of the specified CD/DVD drive: Status Meaning not ready The drive is not ready to be accessed, perhaps due to being engaged in a write operation. Known limitation: "not ready" also occurs when the drive contains a DVD rather than a CD. open The drive contains no disc, or the tray is ejected. playing The drive is playing a disc. paused The previously playing audio or video is now paused. seeking The drive is seeking. stopped The drive contains a CD but is not currently accessing it. Error Handling An exception is thrown on failure. Remarks This function will probably not work on a network drive or non-CD/DVD drive; if it fails in such cases or for any other reason, an exception is thrown. If the tray was recently closed, there may be a delay before the function completes. Related DriveEject, Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveGetType - Syntax & Usage | AutoHotkey v2 DriveGetType Returns the type of the drive which contains the specified path. Type := DriveGetType(Path) Parameters Path Type: String Any path contained by the drive (might also work on UNC paths and mapped drives). Return Value Type: String This function returns the type of the drive which contains the specified path: Unknown, Removable, Fixed, Network, CDROM, or RAMDisk. If the path is invalid (for example, because the drive does not exist), the return value is an empty string. Remarks In general, Path can be any path. Since NTFS supports mounted volumes and directory junctions, different paths with the same drive letter can produce different results. Related Drive functions Examples See example #1 on the Drive Functions page for a demonstration of this function. DriveLock - Syntax & Usage | AutoHotkey v2 DriveLock Prevents the eject feature of the specified drive from working. DriveLock Drive Parameters Drive Type: String The drive letter followed by a colon and an optional backslash (might also work on UNC paths and mapped drives). Error Handling An exception is thrown on failure, such as if the specified drive does not exist or does not support the locking feature. Remarks Most drives cannot be "locked open". However, locking the drive while it is open will probably result in it becoming locked the moment it is closed. This function has no effect on drives that do not support locking (such as most read-only drives). To unlock a drive, call DriveUnlock. If a drive is locked by a script and that script exits, the drive will stay locked until another script or program unlocks it, or the system is restarted. Related DriveUnlock, Drive functions Examples Locks the D drive. DriveLock "D:" DriveSetLabel - Syntax & Usage | AutoHotkey v2 DriveSetLabel Changes the volume label of the specified drive. DriveSetLabel Drive , NewLabel Parameters Drive Type: String The drive letter followed by a colon and an optional backslash (might also work on UNC paths and mapped drives). NewLabel Type: String The new label to set. If omitted, the drive will have no label. Error Handling An exception is thrown on failure. Related DriveGetLabel, Drive functions Examples Changes the volume label of the C drive. DriveSetLabel "C:", "Seagate200" DriveUnlock - Syntax & Usage | AutoHotkey v2 DriveUnlock Restores the eject feature of the specified drive. DriveUnlock Drive Parameters Drive Type: String The drive letter followed by a colon and an optional backslash (might also work on UNC paths and mapped drives). Error Handling An exception is thrown on failure. Remarks This function needs to be called multiple times if the drive was locked multiple times (at least for some drives). For example, if DriveLock("D:") was called three times, DriveUnlock("D:") might need to be called three times to unlock it. Because of this and the fact that there is no way to determine whether a drive is currently locked, it is often useful to keep track of its lock-state in a variable. Related DriveLock, Drive functions Examples Unlocks the D drive. DriveUnlock "D:" Edit - Syntax & Usage | AutoHotkey v2 Edit Opens the current script for editing in the associated editor. Edit The Edit function opens the current script for editing using the associated "edit" verb in the registry (or Notepad if no verb). However, if an editor window appears to have the script already open (based on its window title), that window is activated instead of opening a new instance of the editor. The default program, script or command line executed by the "edit" verb can be changed via Editor settings in the Dash. This function has no effect when operating from within a compiled script. On a related note, AutoHotkey syntax highlighting can be enabled for various editors. In addition, context sensitive help for AutoHotkey functions can be enabled in any editor via this example. Finally, your productivity may be improved by using an auto-completion utility like the script by boiler or the script by Helgef, which works in almost any editor. It watches what you type and displays menus and parameter lists, which does some of the typing for you and reminds you of the order of parameters. Related Reload, How to edit a script, Editors with AutoHotkey support Examples Opens the script for editing. Edit If your editor's command-line usage is something like Editor.exe "Full path of script.ahk", the following can be used to set it as the default editor for ahk files. When you run the script, it will prompt you to select the executable file of your editor. Editor := FileSelect(2,, "Select your editor", "Programs (*.exe)") if Editor = "" ExitApp RegWrite Format('"{1}" "%L"', Editor), "REG_SZ", "HKCR\AutoHotkeyScript\Shell\Edit\Command" EditGetCurrentCol - Syntax & Usage | AutoHotkey v2 EditGetCurrentCol Returns the column number in an Edit control where the caret (text insertion point) resides. CurrentCol := EditGetCurrentCol(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the column number in an Edit control where the caret (text insertion point) resides. The first column is 1. If there is text selected in the control, the return value is the column number where the selection begins. Error Handling An exception is thrown on failure, or if the window or control could not be found. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related EditGetCurrentLine, EditGetLineCount, EditGetLine, EditGetSelectedText, EditPaste, Control functions EditGetCurrentLine - Syntax & Usage | AutoHotkey v2 EditGetCurrentLine Returns the line number in an Edit control where the caret (text insert point) resides. CurrentLine := EditGetCurrentLine (Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the line number in an Edit control where the caret (text insertion point) resides. The first line is 1. If there is text selected in the control, the return value is the line number where the selection begins. Error Handling An exception is thrown on failure, or if the window or control could not be found. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related EditGetCurrentCol, EditGetLineCount, EditGetLine, EditGetSelectedText, EditPaste, Control functions EditGetLine - Syntax & Usage | AutoHotkey v2 EditGetLine Returns the text of the specified line in an Edit control. Line := EditGetLine(N, Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters N Type: Integer The line number. Line 1 is the first line. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the text of line N in an Edit control. Depending on the nature of the control, the string might end in a carriage return (`r) or a carriage return + linefeed (`r`n). Error Handling A TargetError is thrown if the window or control could not be found. A ValueError is thrown if N is out of range or otherwise invalid. An OSError is thrown if a message could not be sent to the control. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related EditGetCurrentCol, EditGetCurrentLine, EditGetLineCount, EditGetSelectedText, EditPaste, Control functions Examples Retrieves the first line of the Notepad's Edit control. line1 := EditGetLine(1, "Edit1", "ahk_class Notepad") EditGetLineCount - Syntax & Usage | AutoHotkey v2 EditGetLineCount Returns the number of lines in an Edit

control. LineCount := EditGetLineCount(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the number of lines in an Edit control. All Edit controls have at least 1 line, even if the control is empty. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related EditGetCurrentCol, EditGetCurrentLine, EditGetLine, EditGetSelectedText, EditPaste, Control functions EditGetSelectedText - Syntax & Usage | AutoHotkey v2 EditGetSelectedText Returns the selected text in an Edit control. Selected := EditGetSelectedText(Control , WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the selected text in an Edit control. If no text is selected, an empty string is returned. Certain types of controls, such as RichEdit20A, might not produce the correct text in some cases (e.g. Metapad). Error Handling A TargetError is thrown if the window or control could not be found. An Error or OSError is thrown if there was a problem retrieving the text. Remarks Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related EditGetCurrentCol, EditGetCurrentLine, EditGetLineCount, EditGetLine, EditPaste, Control functions EditPaste - Syntax & Usage | AutoHotkey v2 EditPaste Pastes the specified string at the caret (text insertion point) in an Edit control. EditPaste String, Control , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters String Type: String The string to paste. Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError may be thrown if a message could not be sent to the control. Remarks The effect is similar to pasting by pressing Ctrl+V, but this function does not affect the contents of the clipboard or require the control to have the keyboard focus. To improve reliability, a delay is done automatically after each use of this function. That delay can be changed via SetControlDelay or by assigning a value to A_ControlDelay. For details, see SetControlDelay remarks. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetText, Control functions Else - Syntax & Usage | AutoHotkey v2 Else Specifies one or more statements to execute if the associated statement's body did not execute. Else Statement Else { Statements } Remarks Every use of an Else must belong to (be associated with) an If statement, Catch, For, Loop or While statement above it. An Else always belongs to the nearest applicable unclaimed statement above it unless a block is used to change that behavior. The condition for an Else statement executing depends on the associated statement: If expression: The expression evaluated to false. For, Loop (any kind), While: The loop had zero iterations. Loop Read: As above, but the presence of Else also prevents an error from being thrown if the file or path is not found. Therefore, Else executes if the file is empty or does not exist. Try...Catch: No exception was thrown within the try block. An Else can be followed immediately by any other single statement on the same line. This is most often used for "else if" ladders (see examples at the bottom). If an Else owns more than one line, those lines must be enclosed in braces (to create a block). However, if only one line belongs to an Else, the braces are optional. For example: if (count > 0) ; No braces are required around the next line because it's only a single line. MsgBox "Press OK to begin the process." else ; Braces must be used around the section below because it consists of more than one line. { WinClose "Untitled - Notepad" MsgBox "There are no items present." } The One True Brace (OTB) style may optionally be used around an Else. For example: if IsDone { ; ... } else if (x < y) { ; ... } else { ; ... } Related Blocks, If, Control Flow Statements Examples Common usage of an Else statement. This example is executed as follows: If Notepad exists: Activate it Send the string "This is a test." followed by Enter. Otherwise (that is, if Notepad does not exist): Activate another window Left-click at the coordinates 100, 200 if WinExist("Untitled - Notepad") { WinActivate Send "This is a test.{Enter}" } else { WinActivate "Some Other Window" MouseClick "Left", 100, 200 } Demonstrates different styles of how the Else statement can be used too. if (x = 1) firstFunction() else if (x = 2) ; "else if" style secondFunction() else if x = 3 { thirdFunction() Sleep 1 } else defaultFunction() ; i.e. Any single statement can be on the same line with an Else. Executes some code if a loop had zero iterations. ; Show File/Internet Explorer windows/tabs. for window in ComObject("Shell.Application").Windows MsgBox "Window #" A_Index ": " window.LocationName else MsgBox "No shell windows found." Enumerator Object - Definition & Usage | AutoHotkey v2 Enumerator Object An enumerator is a type of function object which is called repeatedly to enumerate a sequence of values. Enumerators exist primarily to support For-loops, and are not usually called directly. The for-loop documentation details the process by which an enumerator is called. The script may implement an enumerator to control which values are assigned to the for-loop's variables on each iteration of the loop. Built-in enumerators are instances of the Enumerator class (which is derived from Func), but any function object can potentially be used with a for-loop. Table of Contents Methods: Call: Retrieves the next item or items in an enumeration. Related Methods Call Retrieves the next item or items in an enumeration. Enum.Call(&OutputVar1 , &OutputVar2) EnumFunction(&OutputVar1 , &OutputVar2) &OutputVar1, &OutputVar2 Type: VarRef One or more references to output variables for the enumerator to assign values. Returns a non-zero integer if successful or zero if there were no items remaining. A simple function definition can be used to create an enumerator; in that case, the Call method is implied. When defining your own enumerator, the number of parameters should match the number of variables expected to be passed to the for-loop (before the "in" keyword). This is usually either 1 or 2, but a for-loop can accept up to 19 variables. To allow the method to accept a varying number of variables, declare optional parameters. An exception is thrown when the for-loop attempts to call the method if there are more variables than parameters (too many parameters passed, too few defined) or fewer variables than mandatory parameters. Related For-loop, OwnProps, __Enum (Array), __Enum (Map) EnvGet - Syntax & Usage | AutoHotkey v2 EnvGet Retrieves an environment variable. Value := EnvGet(EnvVarName) Parameters EnvVarName Type: String The name of the environment variable to retrieve. For example: Path := EnvGet("Path"). Return Value Type: String This function returns the value of the specified environment variable. If the specified environment variable is empty or does not exist, an empty string is returned. Remarks The operating system limits each environment variable to 32 KB of text. This function exists because normal script variables are not stored in the environment. This is because performance would be worse and also because the OS limits environment variables to 32 KB. Related EnvSet, environment variables, Run, RunWait Examples Retrieves an environment variable and stores its value in LogonServer. LogonServer := EnvGet("LogonServer") Retrieves and reports the path of the "Program Files" directory. See RegRead example #2 for an alternative method. ProgramFilesDir := EnvGet(A_Is64bitOS ? "ProgramW6432" : "ProgramFiles") MsgBox "Program files are in: " ProgramFilesDir Retrieves and reports the path of the current user's Local AppData directory. LocalAppData := EnvGet ("LocalAppData") MsgBox A_UserName "'s Local directory is located at: " LocalAppData EnvSet - Syntax & Usage | AutoHotkey v2 EnvSet Writes a value to a variable contained in the environment. EnvSet EnvVar , Value Parameters EnvVar Type: String Name of the environment variable to use, e.g. "COMSPEC" or "PATH". Value Type: String Value to set the environment variable to. If omitted, the environment variable is deleted. Error Handling An OSError is thrown on failure. Remarks The operating system limits each environment variable to 32 KB of text. An environment variable created or changed with this function will be accessible only to programs the script launches via Run or RunWait. See environment variables for more details. This function exists because normal script variables are not stored in the environment. This is because performance would be worse and also because the OS limits environment variables to 32 KB. Related EnvGet, environment variables, Run, RunWait Examples Writes some text to the AutGUI variable contained in the environment. EnvSet "AutGUI", "Some text to put in the variable." Error Object | AutoHotkey v2 Error Object class Error extends Object Error objects are thrown by built-in code when a runtime error occurs, and may also be thrown explicitly by the script. Table of Contents Standard Properties Error() Error Types Related Standard Properties Message: An error message. What: What threw the exception. This is usually the name of a function, but is blank for exceptions thrown due to an error in an expression (such as using a math operator on a non-numeric value). Extra: A string value relating to the error, if available. If this value can be converted to a non-empty string, the standard error dialog displays a line with "Specifically:" followed by this string. File: The full path of the script file which contains the line at which the error occurred, or at which the Error object was constructed. Line: The line number at which the error occurred, or at which the Error object was constructed. Stack: A string representing the call stack at the time the Error object was constructed. Each line may be formatted as follows: File (Line) : [What] SourceCode`r`n Represents a call to the function What. File and Line indicate the current script line at this stack depth. SourceCode is an approximation of the source code at that line, as it would be shown in ListLines. > What`r`n Represents the launching of a thread, typically the direct cause of the function call above it. ... N more Indicates that the stack trace was truncated, and there are N more stack entries not shown. Currently the Stack property cannot exceed 2047 characters. Note: The standard error dialog requires Message, Extra, File and Line to be own value properties. Error() Creates an Error object. NewError := Error(Message , What, Extra) Error may be replaced with one of the subclasses listed under Error Types, although some subclasses may take different parameters. The parameters directly correspond to properties described above, but may differ for Error subclasses which override the __New method. Message and Extra are converted to strings. These are displayed by an error dialog if the exception is thrown and not caught. What indicates the source of the error. It can be an arbitrary string, but should be a negative integer or

the name of a running function. Specifying -1 indicates the current function, -2 indicates the function which called it, and so on. If the script is compiled or the value does not identify a valid stack frame, the value is merely converted to a string and assigned to NewError.What. Otherwise, the identified stack frame is used as follows to determine the other properties: NewError.What contains the name of the function. NewError.Line and NewError.File indicate the line which called the function. NewError.Stack contains a partial stack trace, with the indicated stack frame at the top. Use of the What parameter can allow a complex function to use helper functions to perform its work or parameter validation, while omitting those internal details from any reported error information. For example: MyFunction (a, b) { CheckArg "a", a CheckArg "b", b ;... CheckArg(name, value) { if value < 0 throw ValueError(name " is negative", "myfunction", value) } } try MyFunction(1, -1) ; err.Line indicates this line. catch ValueError as err MsgBox Format("{1}: {2}.`n`nFile:`t{3}`nLine:`t{4}`nWhat:`t{5}`nStack:`n{6}" , type(err), err.Message, err.File, err.Line, err.What, err.Stack) try SomeFunction() catch as e MsgBox(type(e) " in " e.What ", which was called at line " e.Line) SomeFunction() { throw Error("Fail", -1) } Error Types The following subclasses of Error are predefined: MemoryError: A memory allocation failed. OSError: An internal function call to a Win32 function failed. Message includes an error code and description generated by the operating system. OSErrors have an additional Number property which contains the error code. Calling OSError(code) with a numeric code sets Number and Message based on the given OS-defined error code. If code is omitted it defaults to A_LastError. TargetError: A function failed because its target could not be found. Message indicates what kind of target, such as a window, control, menu or status bar. TimeoutError: SendMessage timed out. TypeError: An unexpected type of value was used as input for a function, property assignment, or some other operation. Usually Message indicates the expected and actual type, and Extra contains a string representing the errant value. UnsetError: An attempt was made to read the value of a variable, property or item, but there was no value. MemberError PropertyError MethodError UnsetItemError ValueError: An unexpected value was used as input for a function, property assignment, or some other operation. Usually Message indicates which expectation was broken, and Extra contains a string representing the errant value. IndexError: The index parameter of an object's __Item property was invalid or out of range. ZeroDivisionError: Division by zero was attempted in an expression or with the Mod function. Errors are also thrown using the base Error class. Related Throw, Try, Catch, Finally, OnError Exit - Syntax & Usage | AutoHotkey v2 Exit Exits the current thread. Exit ExitCode Parameters ExitCode Type: Integer An integer between -2147483648 and 2147483647 that is returned to its caller when the script exits. This code is accessible to any program that spawned the script, such as another script (via RunWait) or a batch (.bat) file. If omitted, ExitCode defaults to zero. Zero is traditionally used to indicate success. Remarks The Exit function terminates only the current thread. In other words, the stack of functions called directly or indirectly by a menu, timer, or hotkey function will all be returned from as though a Return were immediately encountered in each. If used directly inside such a function -- or in global code -- Exit is equivalent to Return. If the script is not persistent and this is the last thread, the script will terminate after the thread exits. Use ExitApp to completely terminate a script that is persistent. Related ExitApp, OnExit, Functions, Return, Threads, Persistent Examples In this example, the Exit function terminates the call_exit function as well as the calling function. #z:: { call_exit MsgBox "This MsgBox will never happen because of the Exit." call_exit() { Exit ; Terminate this function as well as the calling function. } } ExitApp - Syntax & Usage | AutoHotkey v2 ExitApp Terminates the script. ExitApp ExitCode Parameters ExitCode Type: Integer An integer between -2147483648 and 2147483647 that is returned to its caller when the script exits. This code is accessible to any program that spawned the script, such as another script (via RunWait) or a batch (.bat) file. If omitted, ExitCode defaults to zero. Zero is traditionally used to indicate success. Remarks This is equivalent to choosing "Exit" from the script's tray menu or main menu. Any OnExit functions registered by the script are called before the script terminates. If an OnExit function returns a non-zero integer, the script does not terminate; instead, the current thread exits as if Exit was called. Terminating the script is not the same as exiting each thread. For instance, Finally blocks are not executed and __Delete is not called for objects contained by local variables. ExitApp is often unnecessary in scripts which are not persistent. Related Exit, OnExit, Persistent Examples Press a hotkey to terminate the script. #x::ExitApp ; Win+X File Object - Methods & Properties | AutoHotkey v2 File Object class File extends Object Provides an interface for file input/output. FileOpen returns an object of this type. "FileObj" is used below as a placeholder for any File object, as "File" is the class itself. In addition to the methods and property inherited from Object, File objects have the following predefined methods and properties. Table of Contents Methods: Read: Reads a string of characters from the file and advances the file pointer. Write: Writes a string of characters to the file and advances the file pointer. ReadLine: Reads a line of text from the file and advances the file pointer. WriteLine: Writes a string of characters followed by `n or `r`n depending on the flags used to open the file. Advances the file pointer. ReadNumType: Reads a number from the file and advances the file pointer. WriteNumType: Writes a number to the file and advances the file pointer. RawRead: Reads raw binary data from the file into memory and advances the file pointer. RawWrite: Writes raw binary data to the file and advances the file pointer. Seek: Moves the file pointer. If the second parameter is omitted, it is equivalent to FileObj.Pos := Distance. Close: Closes the file, flushes any data in the cache to disk and releases the share locks. Properties: Pos: Retrieves or sets the position of the file pointer. Length: Retrieves or sets the size of the file. AtEOF: Retrieves a non-zero value if the file pointer has reached the end of the file. Encoding: Retrieves or sets the text encoding used by this file object. Handle: Retrieves a system file handle, intended for use with DllCall. Methods Read Reads a string of characters from the file and advances the file pointer. String := FileObj.Read(Characters) Characters Type: Integer The maximum number of characters to read. If omitted, the rest of the file is read and returned as one string. If the File object was created from a handle to a non-seeking device such as a console buffer or pipe, omitting this parameter may cause the method to fail or return only what data is currently available. Returns a string. Write Writes a string of characters to the file and advances the file pointer. FileObj.Write(String) String Type: String A string to write. Returns the number of bytes (not characters) that were written. ReadLine Reads a line of text from the file and advances the file pointer. TextLine := FileObj.ReadLine() Returns a line of text, excluding the line ending. Lines up to 65,534 characters long can be read. If the length of a line exceeds this, the remainder of the line is returned by subsequent calls to this method. WriteLine Writes a string of characters followed by `n or `r`n depending on the flags used to open the file. Advances the file pointer. FileObj.WriteLine(String) String Type: String An optional string to write. Returns the number of bytes (not characters) that were written. ReadNumType Reads a number from the file and advances the file pointer. Num := FileObj.ReadNumType() NumType is either UInt, Int, Int64, Short, UShort, Char, UChar, Double, or Float. These type names have the same meanings as with DllCall. Returns a number if the read was successful, otherwise an empty string. If the number of bytes read is non-zero but less than the size of NumType, the missing bytes are assumed to be zero. WriteNumType Writes a number to the file and advances the file pointer. FileObj.WriteNumType(Num) Num Type: Integer or Float A number to write. NumType is either UInt, Int, Int64, Short, UShort, Char, UChar, Double, or Float. These type names have the same meanings as with DllCall. Returns the number of bytes that were written. For instance, WriteUInt returns 4 if successful. RawRead Reads raw binary data from the file into memory and advances the file pointer. FileObj.RawRead(Buffer , Bytes) Buffer Type: Object or Integer The Buffer-like object or memory address which will receive the data. Reading into a Buffer is recommended. If Bytes is omitted, it defaults to the size of the buffer. An exception is thrown if Bytes exceeds the size of the buffer. If a memory address is passed, Bytes must also be specified. Bytes Type: Integer The maximum number of bytes to read. This is optional when Buffer is an object; otherwise, it is required. Returns the number of bytes that were read. RawWrite Writes raw binary data to the file and advances the file pointer. FileObj.RawWrite(Data , Bytes) Data Type: Object, String or Integer A Buffer-like object or string containing binary data, or a memory address. If an object or string is specified, Bytes is optional and defaults to the size of the buffer or string. Otherwise, Bytes must also be specified. Bytes Type: Integer The number of bytes to write. This is optional when Data is an object or string; otherwise, it is required. Returns the number of bytes that were written. Seek Moves the file pointer. FileObj.Seek(Distance , Origin) Distance Type: Integer Distance to move, in bytes. Lower values are closer to the beginning of the file. Origin Type: Integer Starting point for the file pointer move. Must be one of the following: 0 (SEEK_SET): Beginning of the file. Distance must be zero or greater. 1 (SEEK_CUR): Current position of the file pointer. 2 (SEEK_END): End of the file. Distance should usually be negative. If omitted, Origin defaults to SEEK_END when Distance is negative and SEEK_SET otherwise. Returns a non-zero value if successful, otherwise zero. Close Closes the file, flushes any data in the cache to disk and releases the share locks. FileObj.Close() Although the file is closed automatically when the object is freed, it is recommended to close the file as soon as possible. Properties Pos Retrieves or sets the current position of the file pointer. Pos := FileObj.Pos FileObj.Pos := NewPos Equivalent to FileObj.Seek (NewPos). The position is a byte offset from the beginning of the file, where 0 is the first byte. When data is written to or read from the file, the file pointer automatically moves to the next byte after that data. Length Retrieves or sets the size of the file. FileSize := FileObj.Length FileObj.Length := NewSize FileSize and NewSize is the size of the file, in bytes. This property should be used only with an actual file. If the File object was created from a handle to a pipe, it may return the amount of data currently available in the pipe's internal buffer, but this behaviour is not guaranteed. AtEOF Retrieves a non-zero value if the file pointer has reached the end of the file, otherwise zero. IsAtEOF := FileObj.AtEOF This property should be used only with an actual file. If the File object was created from a handle to a non-seeking device such as a console buffer or pipe, the returned value may not be meaningful, as such devices do not logically have an "end of file". Encoding Retrieves or sets the text encoding used by this file object. RetrievedEncoding := FileObj.Encoding FileObj.Encoding := NewEncoding NewEncoding may be a numeric code page identifier (see Microsoft Docs) or one of the following strings. RetrievedEncoding is one of the following strings: UTF-8: Unicode UTF-8, equivalent to CP65001. UTF-16: Unicode UTF-16 with little endian byte order, equivalent to CP1200. CPnnn: a code page with numeric identifier nnn. RetrievedEncoding is never a value with the -RAW suffix, regardless of how the file was opened or whether it contains a byte order mark (BOM). Setting NewEncoding never causes a BOM to be added or removed, as the BOM is normally written to the file when it is first created. Setting NewEncoding to UTF-8-RAW or UTF-16-RAW is valid, but the -RAW suffix is ignored. This only applies to FileObj.Encoding, not FileOpen. Handle Returns a system file handle, intended for use with DllCall. See CreateFile. FileObj.Handle File objects internally buffer reads or writes. If data has been written into the object's internal buffer, it is committed to disk before the handle is returned. If the buffer contains data read from file, it is discarded and the actual file pointer is reset to the logical position indicated by FileObj.Pos. FileAppend - Syntax & Usage | AutoHotkey v2 FileAppend Writes text or binary data to the end of a file (first creating the file, if necessary). FileAppend Text , Filename, Options Parameters Text Type: String or Object The text or raw binary data to append to the file. Text may include

linefeed characters (`n) to start new lines. In addition, a single long line can be broken up into several shorter ones by means of a continuation section. A Buffer-like object may be passed to append raw binary data. If a file is created, a byte order mark (BOM) is written only if "UTF-8" or "UTF-16" has been specified within Options. The default encoding is ignored and the data contained by the object is written as-is, regardless of Options. Any object which implements Ptr and Size properties may be used. If Text is blank, Filename will be created as an empty file (but if the file already exists, its modification time will be updated). Filename Type: String The name of the file to be appended, which is assumed to be in A_WorkingDir if an absolute path isn't specified. The destination directory must already exist. Standard Output (stdout): Specifying an asterisk (*) for Filename causes Text to be sent to standard output (stdout). Such text can be redirected to a file, piped to another EXE, or captured by fancy text editors. For example, the following would be valid if typed at a command prompt: "%ProgramFiles% \AutoHotkey\AutoHotkey.exe" "My Script.ahk" >"Error Log.txt" However, text sent to stdout will not appear at the command prompt it was launched from. This can be worked around by 1) compiling the script with the Ahk2Exe ConsoleApp directive, or 2) piping a script's output to another command or program. For example: "%ProgramFiles%\AutoHotkey\AutoHotkey.exe" "My Script.ahk" |more For /F "tokens=*" %L in ('""%ProgramFiles% \AutoHotkey\AutoHotkey.exe" "My Script .ahk""') do @Echo %L Specifying two asterisks (**) for Filename causes Text to be sent to the standard error stream (stderr). Options Type: String Zero or more of the following strings. Separate each option from the next with a single space or tab. For example: "`n UTF-8" Encoding: Specify any of the encoding names accepted by FileEncoding (excluding the empty string) to use that encoding if the file lacks a UTF-8 or UTF-16 byte order mark. If omitted, it defaults to A_FileEncoding (unless Text is an object, in which case no byte order mark is written). RAW: Specify the word RAW (case-insensitive) to write the exact bytes contained by Text to the file as-is, without any conversion. This option overrides any previously specified encoding and vice versa. If Text is not an object, the data size is always a multiple of 2 bytes due to the use of UTF-16 strings. `n (a linefeed character): Inserts a carriage return (`r) before each linefeed (`n) if one is not already present. In other words, it translates from `n to `r`n. This translation typically does not affect performance. If this option is not used, line endings within Text are not changed. Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks To overwrite an existing file, delete it with FileDelete prior to using FileAppend. The target file is automatically closed after the text is appended (except when FileAppend is used in its single-parameter mode inside a file-reading/writing loop). FileOpen in append mode provides more control than FileAppend and allows the file to be kept open rather than opening and closing it each time. Once a file is opened in append mode, use file.Write (string) to append the string. File objects also support binary I/O via RawWrite/RawRead or WriteNum/ReadNum. Related FileOpen/File Object, FileRead, file-reading loop, IniWrite, FileDelete, OutputDebug, continuation sections Examples Creates a file, if necessary, and appends a line. FileAppend "Another line.`n", "C:\My Documents\Test.txt" Use a continuation section to enhance readability and maintainability. FileAppend " (A line of text. By default, the hard carriage return (Enter) between the previous line and this one will be written to the file. This line is indented with a tab; by default, that tab will also be written to the file.)", A_Desktop "\My File.txt" Demonstrates how to automate FTP uploading using the operating system's built-in FTP command. FTPCommandFile := A_ScriptDir "\FTPCommands.txt" FTPLogFile := A_ScriptDir "\FTPLog.txt" try FileDelete FTPCommandFile ; In case previous run was terminated prematurely. FileAppend ("open host.domain.com username password binary cd htdocs put " VarContainingNameOfTargetFile " delete SomeOtherFile.htm rename OldFileName.htm NewFileName.htm ls -l quit"), FTPCommandFile RunWait Format('{1} /c ftp.exe -s:"{2}" >"{3}"', A_ComSpec, FTPCommandFile, FTPLogFile) FileDelete FTPCommandFile ; Delete for security reasons. Run FTPLogFile ; Display the log for review. FileCopy - Syntax & Usage | AutoHotkey v2 FileCopy Copies one or more files. FileCopy SourcePattern, DestPattern , Overwrite Parameters SourcePattern Type: String The name of a single file or folder, or a wildcard pattern such as C:\Temp*.tmp. SourcePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. DestPattern Type: String The name or pattern of the destination, which is assumed to be in A_WorkingDir if an absolute path isn't specified. If present, the first asterisk (*) in the filename is replaced with the source filename excluding its extension, while the first asterisk after the last full stop (.) is replaced with the source file's extension. If an asterisk is present but the extension is omitted, the source file's extension is used. To perform a simple copy -- retaining the existing file name(s) -- specify only the folder name as shown in these mostly equivalent examples: FileCopy "C:*.txt", "C:\My Folder" FileCopy "C:*.txt", "C:\My Folder*.*" The destination directory must already exist. If My Folder does not exist, the first example above will use "My Folder" as the target filename, while the second example will copy no files. Overwrite Type: Integer This parameter determines whether to overwrite files if they already exist. If this parameter is 1 (true), the function overwrites existing files. If omitted or 0 (false), the function does not overwrite existing files. Other values are reserved for future use. Error Handling An Error is thrown if any files failed to be copied, with its Extra property set to the number of failures. If no files were found, an error is thrown only if SourcePattern lacks the wildcards * and ?. In other words, copying a wildcard pattern such as "*.txt" is considered a success when it does not match any files. Unlike FileMove, copying a file onto itself is always counted as an error, even if the overwrite mode is in effect. If files were found, A_LastError is set to 0 (zero) or the result of the operating system's GetLastError() function immediately after the last failure. Otherwise A_LastError contains an error code that might indicate why no files were found. Remarks FileCopy copies files only. To instead copy the contents of a folder (all its files and subfolders), see the examples section below. To copy a single folder (including its subfolders), use DirCopy. The operation will continue even if error(s) are encountered. Related FileMove, DirCopy, DirMove, FileDelete Examples Makes a copy but keep the original file name. FileCopy "C:\My Documents\List1.txt", "D:\Main Backup\" Copies a file into the same directory by providing a new name. FileCopy "C:\My File.txt", "C:\My File New.txt" Copies text files to a new location and gives them a new extension. FileCopy "C:\Folder1*.txt", "D:\New Folder*.bkp" Copies all files and folders inside a folder to a different folder. ErrorCount := CopyFilesAndFolders("C:\My Folder*.*", "D:\Folder to receive all files & folders") if ErrorCount != 0 MsgBox ErrorCount " files/folders could not be copied." CopyFilesAndFolders(SourcePattern, DestinationFolder, DoOverwrite := false) ; Copies all files and folders matching SourcePattern into the folder named DestinationFolder and ; returns the number of files/folders that could not be copied. { ErrorCount := 0 ; First copy all the files (but not the folders): try FileCopy SourcePattern, DestinationFolder, DoOverwrite catch as Err ErrorCount := Err.Extra ; Now copy all the folders: Loop Files, SourcePattern, "D" ; D means "retrieve folders only". { try DirCopy A_LoopFilePath, DestinationFolder "\" A_LoopFileName, DoOverwrite catch { ErrorCount += 1 ; Report each problem folder by name. MsgBox "Could not copy " A_LoopFilePath " into " DestinationFolder } } return ErrorCount } FileCreateShortcut - Syntax & Usage | AutoHotkey v2 FileCreateShortcut Creates a shortcut (.lnk) file. FileCreateShortcut Target, LinkFile , WorkingDir, Args, Description, IconFile, ShortcutKey, IconNumber, RunState Parameters Target Type: String Name of the file that the shortcut refers to, which should include an absolute path unless the file is integrated with the system (e.g. Notepad.exe). The file does not have to exist at the time the shortcut is created; however, if it does not, some systems might alter the path in unexpected ways. LinkFile Type: String Name of the shortcut file to be created, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Be sure to include the .lnk extension. The destination directory must already exist. If the file already exists, it will be overwritten. WorkingDir Type: String Directory that will become Target's current working directory when the shortcut is launched. If blank or omitted, the shortcut will have a blank "Start in" field and the system will provide a default working directory when the shortcut is launched. Args Type: String Parameters that will be passed to Target when it is launched. Separate parameters with spaces. If a parameter contains spaces, enclose it in double quotes. Description Type: String Comments that describe the shortcut (used by the OS to display a tooltip, etc.) IconFile Type: String The full path and name of the icon to be displayed for LinkFile. It must either be an ico file or the very first icon of an EXE or DLL. ShortcutKey Type: String A single letter, number, or the name of a single key from the key list (mouse buttons and other non-standard keys might not be supported). Do not include modifier symbols. Currently, all shortcut keys are created as Ctrl+Alt shortcuts. For example, if the letter B is specified for this parameter, the shortcut key will be Ctrl+Alt+B. IconNumber Type: Integer To use an icon in IconFile other than the first, specify that number here. For example, 2 is the second icon. RunState Type: Integer Launch Target minimized or maximized. If omitted, it defaults to 1 (normal). Otherwise, specify one of the following digits: 1 = Normal 3 = Maximized 7 = Minimized Error Handling An exception is thrown on failure. Remarks Target might not need to include a path if the target file resides in one of the folders listed in the system's PATH environment variable. The ShortcutKey of a newly created shortcut will have no effect unless the shortcut file resides on the desktop or somewhere in the Start Menu. If the ShortcutKey you choose is already in use, your new shortcut takes precedence. An alternative way to create a shortcut to a URL is the following example, which creates a special URL shortcut. Change the first two parameters to suit your preferences: IniWrite "https://www.google.com", "C:\My Shortcut.url", "InternetShortcut", "URL" The following may be optionally added to assign an icon to the above: IniWrite , "C:\My Shortcut.url", "InternetShortcut", "IconFile" IniWrite 0, "C:\My Shortcut.url", "InternetShortcut", "IconIndex" In the above, replace 0 with the index of the icon (0 is used for the first icon) and replace with a URL, EXE, DLL, or ICO file. Examples: "C:\Icons.dll", "C:\App.exe", "https://www.somedomain.com/ShortcutIcon.ico" The operating system will treat a .URL file created by the above as a real shortcut even though it is a plain text file rather than a .LNK file. Related FileGetShortcut, FileAppend Examples The letter "i" in the last parameter makes the shortcut key be Ctrl+Alt+I. FileCreateShortcut "Notepad.exe", A_Desktop "\My Shortcut.lnk", "C:\", A_ScriptFullPath, "My Description", "C:\My Icon.ico", "i" FileDelete - Syntax & Usage | AutoHotkey v2 FileDelete Deletes one or more files. FileDelete FilePattern Parameters FilePattern Type: String The name of a single file or a wildcard pattern such as "C:\Temp*.tmp". FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. To remove an entire folder, along with all its sub-folders and files, use DirDelete. Error Handling An Error is thrown if any files failed to be deleted, with its Extra property set to the number of failures. Deleting a wildcard pattern such as "*.tmp" is considered a success even if it does not match any files. If files were found, A_LastError is set to 0 (zero) or the result of the operating system's GetLastError() function immediately after the last failure. Otherwise A_LastError contains an error code that might indicate why no files were found. Remarks To delete a read-only file, first remove the read-only attribute. For example: FileSetAttrib "-R", "C:\My File.txt". Related FileRecycle, DirDelete, FileCopy, FileMove Examples Deletes all .tmp files in a directory. FileDelete "C:\temp files*.tmp" FileEncoding - Syntax & Usage | AutoHotkey v2 FileEncoding Sets the default encoding for FileRead, Loop Read, FileAppend, and FileOpen. FileEncoding Encoding Parameters Encoding Type: String or Integer One of the following values: UTF-8: Unicode UTF-8, equivalent to CP65001. UTF-8-RAW: As above, but no byte order mark is written when a

new file is created. UTF-16: Unicode UTF-16 with little endian byte order, equivalent to CP1200. UTF-16-RAW: As above, but no byte order mark is written when a new file is created. CPnnn: A code page with numeric identifier nnn. See Code Page Identifiers. nnn: A numeric code page identifier. Return Value Type: String This function returns the previous setting. Remarks The built-in variable A_FileEncoding contains the current setting. If not otherwise set by the script, the default is CP0. CP0 does not universally identify a single code page; rather, it corresponds to the system default ANSI code page, which depends on the system locale or "language for non-Unicode programs" system setting. If CP0 is the current setting, A_FileEncoding returns CP0 (never a blank value). To get the actual code page number, call DllCall("GetACP"). Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. The default encoding is not used if a UTF-8 or UTF-16 byte order mark is present in the file, unless the file is being opened with write-only access (i.e. the previous contents of the file are being discarded). Related FileOpen, StrGet, StrPut, Reading Binary Data FileExist - Syntax & Usage | AutoHotkey v2 FileExist Checks for the existence of a file or folder and returns its attributes. AttributeString := FileExist(FilePattern) Parameters FilePattern Type: String The path, filename, or file pattern to check. FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. Return Value Type: String This function returns the attributes of the first matching file or folder. This string is a subset of RASHNDOCT, where each letter means the following: R = READONLY A = ARCHIVE S = SYSTEM H = HIDDEN N = NORMAL D = DIRECTORY O = OFFLINE C = COMPRESSED T = TEMPORARY L = REPARSE_POINT (typically a symbolic link) If the file has no attributes (rare), "X" is returned. If no file or folder is found, an empty string is returned. Remarks Note that a wildcard check like InStr(FileExist("FolderWithFilesAndSubfolders*"), "D") only tells you whether the first matching file is a folder, not whether a folder exists. To check for the latter, use DirExist. For example: DirExist ("FolderWithFilesAndSubfolders*") Unlike FileGetAttrib, FileExist supports wildcard patterns and always returns a non-empty value if a matching file exists. Since an empty string is seen as "false", the function's return value can always be used as a quasi-boolean value. For example, the statement if FileExist("C:\My File.txt") would be true if the file exists and false otherwise. Since FilePattern may contain wildcards, FileExist may be unsuitable for validating a file path which is to be used with another function or program. For example, FileExist("*.txt") may return attributes even though "*.txt" is not a valid filename. In such cases, FileGetAttrib is preferred. Related DirExist, FileGetAttrib, File-loops Examples Shows a message box if the D drive does exist. if FileExist("D:\") MsgBox "The drive exists." Shows a message box if at least one text file does exist in a directory. if FileExist("D:\Docs*.txt") MsgBox "At least one .txt file exists." Shows a message box if a file does not exist. if not FileExist("C:\Temp\FlagFile.txt") MsgBox "The target file does not exist." Demonstrates how to check a file for a specific attribute. if InStr(FileExist("C:\My File.txt"), "H") MsgBox "The file is hidden." FileGetAttrib - Syntax & Usage | AutoHotkey v2 FileGetAttrib Reports whether a file or folder is read-only, hidden, etc. AttributeString := FileGetAttrib(Filename) Parameters Filename Type: String The name of the target file, which is assumed to be in A_WorkingDir if an absolute path isn't specified. If omitted, the current file of the innermost enclosing File-Loop will be used instead. Unlike FileExist and DirExist, this must be a true filename, not a pattern. Return Value Type: String This function returns the attributes of the file or folder. This string is a subset of RASHNDOCTL, where each letter means the following: R = READONLY A = ARCHIVE S = SYSTEM H = HIDDEN N = NORMAL D = DIRECTORY O = OFFLINE C = COMPRESSED T = TEMPORARY L = REPARSE_POINT (typically a symbolic link) Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks On a related note, to retrieve a file's 8.3 short name, follow this example: Loop Files, "C:\My Documents\Address List.txt" ShortPathName := A_LoopFileShortPath ; Will yield something similar to C:\MYDOCU~1 \ADDRES~1.txt A similar method can be used to get the long name of an 8.3 short name. Related FileExist, DirExist, FileSetAttrib, FileGetTime, FileSetTime, FileGetSize, FileGetVersion, File-loop Examples Stores the attribute letters of a directory in OutputVar. Note that existing directories always have the attribute letter D. OutputVar := FileGetAttrib("C:\New Folder") Check if a particular attribute (the Hidden attribute) is present in the retrieved string. if InStr (FileGetAttrib("C:\My File.txt"), "H") MsgBox "The file is hidden." FileGetShortcut - Syntax & Usage | AutoHotkey v2 FileGetShortcut Retrieves information about a shortcut (.lnk) file, such as its target file. FileGetShortcut LinkFile , &OutTarget, &OutDir, &OutArgs, &OutDescription, &OutIcon, &OutIconNum, &OutRunState Parameters LinkFile Type: String Name of the shortcut file to be analyzed, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Be sure to include the .lnk extension. &OutTarget Type: VarRef A reference to the output variable in which to store the shortcut's target (not including any arguments it might have). For example: C:\WINDOWS\system32\notepad.exe &OutDir Type: VarRef A reference to the output variable in which to store the shortcut's working directory. For example: C:\My Documents. If environment variables such as %WinDir% are present in the string, one way to resolve them is via StrReplace. For example: OutDir := StrReplace(OutDir, "%WinDir%", A_WinDir) &OutArgs Type: VarRef A reference to the output variable in which to store the shortcut's parameters (blank if none). &OutDescription Type: VarRef A reference to the output variable in which to store the shortcut's comments (blank if none). &OutIcon Type: VarRef A reference to the output variable in which to store the filename of the shortcut's icon (blank if none). &OutIconNum Type: VarRef A reference to the output variable in which to store the shortcut's icon number within the icon file (blank if none). This value is most often 1, which means the first icon. &OutRunState Type: VarRef A reference to the output variable in which to store the shortcut's initial launch state, which is one of the following digits: 1 = Normal 3 = Maximized 7 = Minimized Error Handling An OSError is thrown on failure. Remarks Any of the output variables may be omitted if the corresponding information is not needed. Related FileCreateShortcut, SplitPath Examples Allows the user to select an .lnk file to show its information. LinkFile := FileSelect(32,, "Pick a shortcut to analyze.", "Shortcuts (*.lnk)") if LinkFile = "" return FileGetShortcut LinkFile, &OutTarget, &OutDir, &OutArgs, &OutDesc, &OutIcon, &OutIconNum, &OutRunState MsgBox OutTarget "`n" OutDir "`n" OutArgs "`n" OutDesc "`n" OutIcon "`n" OutIconNum "`n" OutRunState FileGetSize - Syntax & Usage | AutoHotkey v2 FileGetSize Retrieves the size of a file. Size := FileGetSize(Filename, Units) Parameters Filename Type: String The name of the target file, which is assumed to be in A_WorkingDir if an absolute path isn't specified. If omitted, the current file of the innermost enclosing File-Loop will be used instead. Units Type: String If present, this parameter causes the result to be returned in units other than bytes: K = kilobytes M = megabytes Return Value Type: Integer This function returns the size of the specified file (rounded down to the nearest whole number). Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks Files of any size are supported, even those over 4 gigabytes, and even if Units is bytes. If the target file is a directory, the size will be reported as whatever the OS believes it to be (probably zero in all cases). To calculate the size of folder, including all its files, follow this example: FolderSize := 0 WhichFolder := DirSelect() ; Ask the user to pick a folder. Loop Files, WhichFolder "*.*", "R" FolderSize += A_LoopFileSize MsgBox "Size of " WhichFolder " is " FolderSize " bytes." Related FileGetAttrib, FileSetAttrib, FileGetTime, FileSetTime, FileGetVersion, File-loop Examples Retrieves the size in bytes and stores it in Size. Size := FileGetSize("C:\My Documents\test.doc") Retrieves the size in kilobytes and stores it in Size. Size := FileGetSize("C:\My Documents\test.doc", "K") FileGetTime - Syntax & Usage | AutoHotkey v2 FileGetTime Retrieves the datetime stamp of a file or folder. Timestamp := FileGetTime(Filename, WhichTime) Parameters Filename Type: String The name of the target file or folder, which is assumed to be in A_WorkingDir if an absolute path isn't specified. If omitted, the current file of the innermost enclosing File-Loop will be used instead. WhichTime Type: String If blank or omitted, it defaults to M (modification time). Otherwise, specify one of the following letters to set which timestamp should be retrieved: M = Modification time C = Creation time A = Last access time Return Value Type: String This function returns a string of digits in the YYYYMMDDHH24MISS format. The time is your own local time, not UTC/GMT. This string should not be treated as a number (one should not perform math on it or compare it numerically). Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks See YYYYMMDDHH24MISS for an explanation of dates and times. Related FileSetTime, FormatTime, FileGetAttrib, FileSetAttrib, FileGetSize, FileGetVersion, File-loop, DateAdd, DateDiff Examples Retrieves the modification time and stores it in Timestamp. Timestamp := FileGetTime("C:\My Documents\test.doc") Retrieves the creation time and stores it in Timestamp. Timestamp := FileGetTime("C:\My Documents\test.doc", "C") FileGetVersion - Syntax & Usage | AutoHotkey v2 FileGetVersion Retrieves the version of a file. Version := FileGetVersion(Filename) Parameters Filename Type: String The name of the target file. If a full path is not specified, this function uses the search sequence specified by the system LoadLibrary function. If omitted, the current file of the innermost enclosing File-Loop will be used instead. Return Value Type: String This function returns the version number of the specified file. Error Handling An OSError is thrown on failure, such as if the file lacks version information. A_LastError is set to the result of the operating system's GetLastError() function. Remarks Most non-executable files (and even some EXEs) have no version, and thus an error will be thrown. Related FileGetAttrib, FileSetAttrib, FileGetTime, FileSetTime, FileGetSize, File-loop Examples Retrieves the version of a file and stores it in Version. Version := FileGetVersion("C:\My Application.exe") Retrieves the version of the file "AutoHotkey.exe" located in AutoHotkey's installation directory and stores it in Version. Version := FileGetVersion(A_ProgramFiles "\AutoHotkey\AutoHotkey.exe") FileInstall - Syntax & Usage | AutoHotkey v2 FileInstall Includes the specified file inside the compiled version of the script. FileInstall Source, Dest , Overwrite Parameters Source Type: String The name of a single file to be added to the compiled EXE. The file is assumed to be in (or relative to) the script's own directory if an absolute path isn't specified. This parameter must be a quoted literal string (not a variable or any other expression), and must be listed to the right of the function name FileInstall (that is, not on a continuation line beneath it). Dest Type: String When Source is extracted from the EXE, this is the name of the file to be created. It is assumed to be in A_WorkingDir if an absolute path isn't specified. The destination directory must already exist. Overwrite Type: Integer This parameter determines whether to overwrite files if they already exist. If this parameter is 1 (true), the function overwrites existing files. If omitted or 0 (false), the function does not overwrite existing files. Other values are reserved for future use. Error Handling An exception is thrown on failure. Any case where the file cannot be written to the destination is considered failure. For example: The destination file already exists and the Overwrite parameter was 0 (false) or omitted. The destination file could not be opened due to a permission error, or because the file is in use. The destination path was invalid or specifies a folder which does not exist. The source file does not exist (only for uncompiled scripts). Remarks When a call to this function is read by Ahk2Exe during compilation of the script, the file specified by Source is added to the resulting compiled script. Later, when the compiled script EXE runs and the call to

FileInstall is executed, the file is extracted from the EXE and written to the location specified by Dest. Files added to a script are neither compressed nor encrypted during compilation, but the compiled script EXE can be compressed by using the appropriate option in Ahk2Exe. If this function is used in a normal (uncompiled) script, a simple file copy will be performed instead -- this helps the testing of scripts that will eventually be compiled. No action is taken if the full source and destination paths are equal, as attempting to copy the file to its current location would result in an error. The paths are compared with lstrcmpi after expansion with GetFullPathName. Related FileCopy, #Include Examples Includes a text file inside the compiled version of the script. Later, when the compiled script is executed, the included file is extracted to another location with another name. If a file with this name already exists at this location, it will be overwritten. FileInstall "My File.txt", A_Desktop "\Example File.txt", 1 FileMove - Syntax & Usage | AutoHotkey v2 FileMove Moves or renames one or more files. FileMove SourcePattern, DestPattern , Overwrite Parameters SourcePattern Type: String The name of a single file or a wildcard pattern such as C:\Temp*.tmp. SourcePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. DestPattern Type: String The name or pattern of the destination, which is assumed to be in A_WorkingDir if an absolute path isn't specified. If present, the first asterisk (*) in the filename is replaced with the source filename excluding its extension, while the first asterisk after the last full stop (.) is replaced with the source file's extension. If an asterisk is present but the extension is omitted, the source file's extension is used. To perform a simple move -- retaining the existing file name(s) -- specify only the folder name as shown in these mostly equivalent examples: FileMove "C:*.txt", "C:\My Folder" FileMove "C:*.txt", "C:\My Folder*.*" The destination directory must already exist. If My Folder does not exist, the first example above will use "My Folder" as the target filename, while the second example will move no files. Overwrite Type: Integer This parameter determines whether to overwrite files if they already exist. If this parameter is 1 (true), the function overwrites existing files. If omitted or 0 (false), the function does not overwrite existing files. Other values are reserved for future use. Error Handling An Error is thrown if any files failed to be moved, with its Extra property set to the number of failures. If no files were found, an exception is thrown only if SourcePattern lacks the wildcards * and ?. In other words, moving a wildcard pattern such as "*.txt" is considered a success when it does not match any files. Unlike FileCopy, moving a file onto itself is always considered successful, even if the overwrite mode is not in effect. If files were found, A_LastError is set to 0 (zero) or the result of the operating system's GetLastError() function immediately after the last failure. Otherwise A_LastError contains an error code that might indicate why no files were found. Remarks FileMove moves files only. To instead move the contents of a folder (all its files and subfolders), see the examples section below. To move or rename a single folder, use DirMove. The operation will continue even if error(s) are encountered. Although this function is capable of moving files to a different volume, the operation will take longer than a same-volume move. This is because a same-volume move is similar to a rename, and therefore much faster. Related FileCopy, DirCopy, DirMove, FileDelete Examples Moves a file without renaming it. FileMove "C:\My Documents\List1.txt", "D:\Main Backup\" Renames a single file. FileMove "C:\File Before.txt", "C:\File After.txt" Moves text files to a new location and gives them a new extension. FileMove "C:\Folder1*.txt", "D:\New Folder*.bkp" Moves all files and folders inside a folder to a different folder. ErrorCount := MoveFilesAndFolders("C:\My Folder*.*", "D:\Folder to receive all files & folders") if ErrorCount != 0 MsgBox ErrorCount " files/folders could not be moved." MoveFilesAndFolders(SourcePattern, DestinationFolder, DoOverwrite := false) ; Moves all files and folders matching SourcePattern into the folder named DestinationFolder and ; returns the number of files/folders that could not be moved. { ErrorCount := 0 if DoOverwrite = 1 DoOverwrite := 2 ; See DirMove for description of mode 2 vs. 1. ; First move all the files (but not the folders): try FileMove SourcePattern, DestinationFolder, DoOverwrite catch as Err ErrorCount := Err.Extra ; Now move all the folders: Loop Files, SourcePattern, "D" ; D means "retrieve folders only". { try DirMove A_LoopFilePath, DestinationFolder "\" A_LoopFileName, DoOverwrite catch { ErrorCount += 1 ; Report each problem folder by name. MsgBox "Could not move " A_LoopFilePath " into " DestinationFolder } } return ErrorCount } FileOpen - Syntax & Usage | AutoHotkey v2 FileOpen Opens a file to read specific content from it and/or to write new content into it. FileObj := FileOpen(Filename, Flags , Encoding) Parameters Filename Type: String The path of the file to open, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Specify an asterisk (or two) as shown below to open the standard input/output/error stream: FileOpen("*", "r") ; for stdin FileOpen("*", "w") ; for stdout FileOpen("**", "w") ; for stderr Flags Type: String or Integer Either a string of characters indicating the desired access mode followed by other options (with optional spaces or tabs in between); or a combination (sum) of numeric flags. Supported values are described in the tables below. Encoding Type: String or Integer The code page to use for text I/O if the file does not contain a UTF-8 or UTF-16 byte order mark, or if the h (handle) flag is used. If omitted, the current value of A_FileEncoding is used. See FileEncoding for a list of possible values. Flags Access modes (mutually-exclusive) Flag Dec Hex Description r 0 0x0 Read: Fails if the file doesn't exist. w 1 0x1 Write: Creates a new file, overwriting any existing file. a 2 0x2 Append: Creates a new file if the file didn't exist, otherwise moves the file pointer to the end of the file. rw 3 0x3 Read/Write: Creates a new file if the file didn't exist. h Indicates that Filename is a file handle to wrap in an object. Sharing mode flags are ignored and the file or stream represented by the handle is not checked for a byte order mark. The file handle is not closed automatically when the file object is destroyed and calling Close has no effect. Note that Seek, Pos and Length should not be used if Filename is a handle to a nonseeking device such as a pipe or a communications device. Sharing mode flags Flag Dec Hex Description - rwd Locks the file for read, write and/or delete access. Any combination of r, w and d may be used. Specifying - is the same as specifying -rwd. If omitted entirely, the default is to share all access. 0 0x0 If Flags is numeric, the absence of sharing mode flags causes the file to be locked. 256 0x100 Shares read access. 512 0x200 Shares write access. 1024 0x400 Shares delete access. End of line (EOL) options Flag Dec Hex Description `n 4 0x4 Replace `r`n with `n when reading and `n with `r`n when writing. `r 8 0x8 Replace standalone `r with `n when reading. Return Value Type: File The return value is a new File object encapsulating the open handle to the file. Use the methods and properties of this object to access the file's contents. Errors If the file cannot be opened, an OSError is thrown. Remarks File.ReadLine always supports `n, `r`n and `r as line endings and does not include them in its return value, regardless of whether the `r or `n options are used. The options only affect translation of line endings within the text returned by File.Read or written by File.Write or File.WriteLine. When a UTF-8 or UTF-16 file is created, a byte order mark is written to the file unless Encoding (or A_FileEncoding if Encoding is omitted) contains UTF-8-RAW or UTF-16-RAW. When a file containing a UTF-8 or UTF-16 byte order mark (BOM) is opened with read access, the BOM is excluded from the output by positioning the file pointer after it. Therefore, File.Pos may report 3 or 2 immediately after opening the file. Related FileEncoding, File Object, FileRead Examples Writes some text to a file then reads it back into memory (it provides the same functionality as this DllCall example). FileName := FileSelect("S16",, "Create a new file:") if (FileName = "") return try FileObj := FileOpen(FileName, "w") catch as Err { MsgBox "Can't open '" FileName "' for writing." . "`n`n" Type(Err) ": " Err.Message return } TestString := "This is a test string.`r`n" ; When writing a file this way, use `r`n rather than `n to start a new line. FileObj.Write(TestString) FileObj.Close() ; Now that the file was written, read its contents back into memory. try FileObj := FileOpen(FileName, "r-d") ; read the file ("r"), share all access except for delete ("-d") catch as Err { MsgBox "Can't open '" FileName "' for reading." . "`n`n" Type(Err) ": " Err.Message return } CharsToRead := StrLen(TestString) TestString := FileObj.Read (CharsToRead) FileObj.Close() MsgBox "The following string was read from the file: " TestString Opens the script in read-only mode and read its first line. Script := FileOpen(A_ScriptFullPath, "r") MsgBox Script.ReadLine() Demonstrates the usage of the standard input/output streams. ; Open a console window for this demonstration: DllCall("AllocConsole") ; Open the application's stdin/stdout streams. stdin := FileOpen("*", "r") stdout := FileOpen("*", "w") stdout.Write ("Enter your query.`n\> ") stdout.Read(0) ; Flush the write buffer. query := RTrim(stdin.ReadLine(), "`n") stdout.WriteLine("Your query was '" query "'. Have a nice day.") stdout.Read(0) ; Flush the write buffer. Sleep 5000 FileRead - Syntax & Usage | AutoHotkey v2 FileRead Retrieves the contents of a file. Text := FileRead(Filename , Options) Parameters Filename Type: String The name of the file to read, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Options Type: String Zero or more of the following strings. Separate each option from the next with a single space or tab. For example: "`n m5000 UTF-8" Encoding: Specify any of the encoding names accepted by FileEncoding (excluding the empty string) to use that encoding if the file lacks a UTF-8 or UTF-16 byte order mark. If omitted, it defaults to A_FileEncoding. RAW: Specify the word RAW (case-insensitive) to read the file's content as raw binary data and return a Buffer object instead of a string. This option overrides any previously specified encoding and vice versa. m1024: If this option is omitted, the entire file is loaded unless there is insufficient memory, in which case an error message is shown and the thread exits (but Try can be used to avoid this). Otherwise, replace 1024 with a decimal or hexadecimal number of bytes. If the file is larger than this, only its leading part is loaded. Note: This might result in the last line ending in a naked carriage return (`r) rather than `r`n. `n (a linefeed character): Replaces any/all occurrences of carriage return & linefeed (`r`n) with linefeed (`n). However, this translation reduces performance and is usually not necessary. For example, text containing `r`n is already in the right format to be added to a Gui Edit control. The following parsing loop will work correctly regardless of whether each line ends in `r`n or just `n: Loop Parse, MyFileContents, "`n", "`r". Return Value Type: String or Buffer This function returns the contents of the specified file. The return value is a Buffer object if the RAW option is in effect and the file can be opened; otherwise, it is a string. If the file does not exist or cannot be opened for any other reason, an empty string is returned. Error Handling An OSError is thrown if there was a problem opening or reading the file. A file greater than 4 GB in size will cause a MemoryError to be thrown unless the *m option is present, in which case the leading part of the file is loaded. A MemoryError will also be thrown if the program is unable to allocate enough memory to contain the requested amount of data. A_LastError is set to the result of the operating system's GetLastError() function. Reading Binary Data When the RAW option is used, the return value is a Buffer object containing the raw, unmodified contents of the file. The object's Size property returns the number of bytes read. NumGet or StrGet can be used to retrieve data from the buffer. For example: buf := FileRead(A_AhkPath, "RAW") if StrGet(buf, 2, "cp0") == "MZ" ; Looks like an executable file... { ; Read machine type from COFF file header. machine := NumGet(buf, NumGet(buf, 0x3C, "uint") + 4, "ushort") machine := machine=0x8664 ? "x64" : machine=0x014C ? "x86" : "unknown" ; Display machine type and file size. MsgBox "This " machine " executable is " buf.Size " bytes." } buf := "" This option is generally required for reading binary data because by default, any bytes read from file are interpreted as text and may be converted from the source file's encoding (as specified in the options or by A_FileEncoding) to the script's native encoding, UTF-16. If the data is not UTF-16 text, this conversion generally changes the data in undesired ways. For another demonstration of the RAW option, see ClipboardAll example #2. Finally, FileOpen and File.RawRead or File.ReadNum may be

used to read binary data without first reading the entire file into memory. Remarks When the goal is to load all or a large part of a file into memory, FileRead performs much better than using a file-reading loop. If there is concern about using too much memory, check the file size beforehand with FileGetSize. FileOpen provides more advanced functionality than FileRead, such as reading or writing data at a specific location in the file without reading the entire file into memory. See File Object for a list of functions. Related FileEncoding, FileOpen/File Object, file-reading loop, FileGetSize, FileAppend, IniRead, Sort, Download Examples Reads a text file into MyText. MyText := FileRead("C:\My Documents\My File.txt") Quickly sorts the contents of a file. Contents := FileRead("C:\Address List.txt") Contents := Sort(Contents) FileDelete "C:\Address List (alphabetical).txt" FileAppend Contents, "C:\Address List (alphabetical).txt" Contents := "" ; Free the memory. FileRecycle - Syntax & Usage | AutoHotkey v2 FileRecycle Sends a file or directory to the recycle bin if possible, or permanently deletes it. FileRecycle FilePattern Parameters FilePattern Type: String The name of a single file or a wildcard pattern such as C:\Temp*.tmp. FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. To recycle an entire directory, provide its name without a trailing backslash. Error Handling An exception is thrown on failure. Remarks SHFileOperation is used to do the actual work. This function may permanently delete the file if it is too large to be recycled; also, a warning should be shown before this occurs. The file may be permanently deleted without warning if the file cannot be recycled for other reasons, such as: The file is on a removable drive. The Recycle Bin has been disabled, such as via the NukeOnDelete registry value. Related FileRecycleEmpty, FileDelete, FileCopy, FileMove Examples Sends all .tmp files in a directory to the recycle bin if possible. FileRecycle "C:\temp files*.tmp" FileRecycleEmpty - Syntax & Usage | AutoHotkey v2 FileRecycleEmpty Empties the recycle bin. FileRecycleEmpty DriveLetter Parameters DriveLetter Type: String If omitted, the recycle bin for all drives is emptied. Otherwise, specify a drive letter such as C:\ Error Handling An OSError is thrown on failure. Related FileRecycle, FileDelete, FileCopy, FileMove Examples Empties the recycle bin of the C drive. FileRecycleEmpty "C:\" FileSelect - Syntax & Usage | AutoHotkey v2 FileSelect Displays a standard dialog that allows the user to open or save file(s). SelectedFile := FileSelect(Options, RootDir\Filename, Title, Filter) Parameters Options Type: String or Integer If omitted, it will default to zero, which is the same as having none of the options below. Otherwise, it can be a number or one of the letters listed below, optionally followed by a number. For example, "M", 1 and "M1" are all valid (but not equivalent). D: Select Folder (Directory). Specify the letter D to allow the user to select a folder rather than a file. The dialog has most of the same features as when selecting a file, but does not support filters (Filter must be omitted or blank). M: Multi-select. Specify the letter M to allow the user to select more than one file via shift-click, control-click, or other means. In this case, the return value is an Array instead of a string. To extract the individual files, see the example at the bottom of this page. S: Save dialog. Specify the letter S to cause the dialog to always contain a Save button instead of an Open button. The following numbers can be used. To put more than one of them into effect, add them up. For example, to use 1 and 2, specify the number 3. 1: File Must Exist 2: Path Must Exist 8: Prompt to Create New File 16: Prompt to Overwrite File 32: Shortcuts (.lnk files) are selected as-is rather than being resolved to their targets. This option also prevents navigation into a folder via a folder shortcut. As the "Prompt to Overwrite" option is supported only by the Save dialog, specifying that option without the "Prompt to Create" option also puts the "S" option into effect. Similarly, the "Prompt to Create" option has no effect when the "S" option is present. Specifying the number 24 enables whichever type of prompt is supported by the dialog. RootDir\Filename Type: String If present, this parameter contains one or both of the following: RootDir: The root (starting) directory, which is assumed to be a subfolder in A_WorkingDir if an absolute path is not specified. If omitted or blank, the starting directory will be a default that might depend on the OS version (it will likely be the directory most recently selected by the user during a prior use of FileSelect). Filename: The default filename to initially show in the dialog's edit field. Only the naked filename (with no path) will be shown. To ensure that the dialog is properly shown, ensure that no illegal characters are present (such as /<|:"). Examples: "C:\My Pictures\Default Image Name.gif" ; Both RootDir and Filename are present. "C:\My Pictures" ; Only RootDir is present. "My Pictures" ; Only RootDir is present, and it's relative to the current working directory. "My File" ; Only Filename is present (but if "My File" exists as a folder, it is assumed to be RootDir). Title Type: String The title of the file-selection window. If omitted or blank, it will default to "Select File - " A_ScriptName (i.e. the name of the current script), unless the "D" option is present, in which case the word "File" is replaced with "Folder". Filter Type: String Indicates which types of files are shown by the dialog. Example: Documents (*.txt) Example: Audio (*.wav; *.mp2; *.mp3) If omitted, the filter defaults to All Files (*.*). Otherwise, the filter uses the indicated string but also provides an option for All Files (*.*) in the dialog's "files of type" drop-down list. To include more than one file extension in the filter, separate them with semicolons as illustrated in the example above. This parameter must be blank or omitted if the "D" option is present. Return Value Type: String or Array of Strings If multi-select is not in effect, this function returns the full path and name of the single file or folder chosen by the user, or an empty string if the user cancels the dialog. If the M option (multi-select) is in effect, this function returns an array of items, where each item is the full path and name of a single file. The example at the bottom of this page demonstrates how to extract the files one by one. If the user cancels the dialog, the array is empty (has zero items). Remarks A file-selection dialog usually looks like this: A GUI window may display a modal file-selection dialog by means of the +OwnDialogs option. A modal dialog prevents the user from interacting with the GUI window until the dialog is dismissed. Known limitation: A timer that launches during the display of a FileSelect dialog will postpone the effect of the user's clicks inside the dialog until after the timer finishes. To work around this, avoid using timers whose subroutines take a long time to finish, or disable all timers during the dialog: Thread "NoTimers" SelectedFile := FileSelect() Thread "NoTimers", false Related DirSelect, MsgBox, InputBox, ToolTip, GUI, CLSID List, parsing loop, SplitPath Also, the operating system offers standard dialog boxes that prompt the user to pick a font, color, or icon. These dialogs can be displayed via DllCall as demonstrated at GitHub. Examples Allows the user to select an existing .txt or .doc file. SelectedFile := FileSelect(3, , "Open a file", "Text Documents (*.txt; *.doc)") if SelectedFile = "" MsgBox "The dialog was canceled." else MsgBox "The following file was selected:`n" SelectedFile Allows the user to select multiple existing files. SelectedFiles := FileSelect("M3") ; M3 = Multiselect existing files. if SelectedFiles.Length = 0 { MsgBox "The dialog was canceled." return } for FileName in SelectedFiles { Result := MsgBox("File #" A_Index " of " SelectedFiles.Length ":`n" FileName "`n`nContinue?",, "YN") if Result = "No" break } Allows the user to select a folder. SelectedFolder := FileSelect("D", , "Select a folder") if SelectedFolder = "" MsgBox "The dialog was canceled." else MsgBox "The following folder was selected:`n" SelectedFolder FileSetAttrib - Syntax & Usage | AutoHotkey v2 FileSetAttrib Changes the attributes of one or more files or folders. Wildcards are supported. FileSetAttrib Attributes , FilePattern, Mode Parameters Attributes Type: String The attributes to change. For example, +HA-R. To easily turn on, turn off or toggle attributes, prefix one or more of the following attribute letters with a plus sign (+), minus sign (-) or caret (^), respectively: R = READONLY A = ARCHIVE S = SYSTEM H = HIDDEN N = NORMAL (this is valid only when used without any other attributes) O = OFFLINE T = TEMPORARY Note: Currently, the compression state of files cannot be changed with this function. FilePattern Type: String The name of a single file or folder, or a wildcard pattern such as "C:\Temp*.tmp". FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. If omitted, the current file of the innermost enclosing File-Loop will be used instead. Mode Type: String If blank or omitted, only files are operated upon and subdirectories are not recursed into. Otherwise, specify zero or more of the following letters: D = Include directories (folders). F = Include files. If both F and D are omitted, files are included but not folders. R = Subfolders are recursed into so that files and folders contained therein are operated upon if they match FilePattern. All subfolders will be recursed into, not just those whose names match FilePattern. If R is omitted, files and folders in subfolders are not included. Error Handling An Error is thrown if any files failed to be changed, with its Extra property set to the number of failures. If files were found, A_LastError is set to 0 (zero) or the result of the operating system's GetLastError() function immediately after the last failure. Otherwise A_LastError contains an error code that might indicate why no files were found. Related FileGetAttrib, FileGetTime, FileSetTime, FileGetSize, FileGetVersion, File-loop Examples Turns on the "read-only" and "hidden" attributes of all files and directories (subdirectories are not recursed into). FileSetAttrib "+RH", "C:\MyFiles*.*", "DF" ; +RH is identical to +R+H Toggles the "hidden" attribute of a single directory. FileSetAttrib "^H", "C:\MyFiles" Turns off the "read-only" attribute and turns on the "archive" attribute of a single file. FileSetAttrib "-R+A", "C:\New Text File.txt" Recurses through all .ini files on the C drive and turns on their "archive" attribute. FileSetAttrib "+A", "C:*.ini", "R" FileSetTime - Syntax & Usage | AutoHotkey v2 FileSetTime Changes the datetime stamp of one or more files or folders. Wildcards are supported. FileSetTime YYYYMMDDHH24MISS, FilePattern, WhichTime, Mode Parameters YYYYMMDDHH24MISS Type: String If blank or omitted, it defaults to the current time. Otherwise, specify the time to use for the operation (see Remarks for the format). Years prior to 1601 are not supported. FilePattern Type: String The name of a single file or folder, or a wildcard pattern such as C:\Temp*.tmp. FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. If omitted, the current file of the innermost enclosing File-Loop will be used instead. WhichTime Type: String If blank or omitted, it defaults to M (modification time). Otherwise, specify one of the following letters to set which timestamp should be changed: M = Modification time C = Creation time A = Last access time Mode Type: String If blank or omitted, only files are operated upon and subdirectories are not recursed into. Otherwise, specify zero or more of the following letters: D = Include directories (folders). F = Include files. If both F and D are omitted, files are included but not folders. R = Subfolders are recursed into so that files and folders contained therein are operated upon if they match FilePattern. All subfolders will be recursed into, not just those whose names match FilePattern. If R is omitted, files and folders in subfolders are not included. Note: If FilePattern is a single folder rather than a wildcard pattern, it will always be operated upon regardless of this setting. Error Handling An Error is thrown if any files failed to be changed, with its Extra property set to the number of failures. If files were found, A_LastError is set to 0 (zero) or the result of the operating system's GetLastError() function immediately after the last failure. Otherwise A_LastError contains an error code that might indicate why no files were found. Remarks A file's last access time might not be as precise on FAT16 & FAT32 volumes as it is on NTFS volumes. The elements of the YYYYMMDDHH24MISS format are: Element Description YYYY The 4-digit year MM The 2-digit month (01-12) DD The 2-digit day of the month (01-31) HH24 The 2-digit hour in 24-hour format (00-23). For example, 09 is 9am and 21 is 9pm. MI The 2-digit minutes (00-59) SS The 2-digit seconds (00-59) If only a partial string is given for YYYYMMDDHH24MISS (e.g. 200403), any remaining element that has been omitted will be supplied with the following default values: MM = Month 01 DD = Day 01 HH24 = Hour 00 MI = Minute 00 SS = Second 00 The built-in variable A_Now contains the current local time in the above format. Similarly, A_NowUTC contains the current Coordinated Universal Time. Note: Date-time values can be compared, added to, or subtracted from via

DateAdd and DateDiff. Also, it is best to not use greater-than or less-than to compare times unless they are both the same string length. This is because they would be compared as numbers; for example, 20040201 is always numerically less (but chronologically greater) than 200401010533. So instead use DateDiff to find out whether the amount of time between them is positive or negative. Related FileGetTime, FileGetAttrib, FileSetAttrib, FileGetSize, FileGetVersion, FormatTime, File-loop, DateAdd, DateDiff Examples Sets the modification time to the current time for all matching files. FileSetTime "", "C:\temp*.txt" Sets the modification date (time will be midnight). FileSetTime 20040122, "C:\My Documents\test.doc" Sets the creation date. The time will be set to 4:55pm. FileSetTime 200401221655, "C:\My Documents\test.doc", "C" Changes the mod-date of all files that match a pattern. Any matching folders will also be changed due to the last parameter. FileSetTime 20040122165500, "C:\Temp*.*", "M", "DF" Finally - Syntax & Usage | AutoHotkey v2 Finally Ensures that one or more statements are always executed after a Try statement finishes. Finally Statement Finally { Statements } Remarks Every use of finally must belong to (be associated with) a try statement above it (after any optional catch and/or else). A finally always belongs to the nearest unclaimed try statement above it unless a block is used to change that behavior. Try statements behave differently depending on whether catch or finally is present. For more information, see Try. Goto, break, continue and return cannot be used to exit a finally block, as that would require suppressing any control flow instructions within the try block. For example, if try uses return 42, the value 42 is returned after the finally block executes. Attempts to jump out of a finally block using one of these statements are detected as errors at load time where possible, or at run time otherwise. Finally statements are not executed if the script is directly terminated by any means, including the tray menu or ExitApp. The One True Brace (OTB) style may optionally be used with the finally statement. For example: try { ... } finally { ... } try { ... } catch { ... } else { ... } finally { ... } Related Try, Catch, Else, Throw, Blocks Examples Demonstrates the behavior of finally in detail. try { ToolTip "Working..." Example1() } catch as e { ; For more detail about the object that e contains, see Error. MsgBox(Type(e) " thrown!`n`nwhat: " e.what "`nfile: " e.file . "`nline: " e.line "`nmessage: " e.message "`nextra: " e.extra,, 16) } finally { ToolTip ; hide the tooltip } MsgBox "Done!" ; This function has a Finally block that acts as cleanup code Example1() { try Example2() finally MsgBox "This is always executed regardless of exceptions" } ; This function fails when the minutes are odd Example2() { if Mod(A_Min, 2) throw Error ("That's odd...") MsgBox "Example2 did not fail" } Float - Syntax & Usage | AutoHotkey v2 Float Converts a numeric string or integer value to a floating-point number. FltValue := Float(Value) Return Value Type: Float This function returns the result of converting Value to a pure floating-point number (having the type name "Float"), or Value itself if it is already the correct type. Remarks If the value cannot be converted, a TypeError is thrown. To determine if a value can be converted to a floating-point number, use the IsNumber function. Float is actually a class, but can be called as a function. Value is Float can be used to check whether a value is a pure floating-point number. Related Type, Integer, Number, String, Values, Expressions, Is functions For Loop - Syntax & Usage | AutoHotkey v2 For-loop Repeats a series of functions once for each key-value pair in an object. For Value1 , Value2 in Expression Parameters Value1 Value2 Type: Variable The variables in which to store the values returned by the enumerator at the beginning of each iteration. The nature of these values is defined by the enumerator, which is determined by Expression. These variables cannot be dynamic. When the loop breaks or completes, these variables are restored to their former values. If a loop variable is a ByRef parameter, the target variable is unaffected by the loop. Closures which reference the variable (if local) are also unaffected and will see only the value it had outside the loop. Note: Even if defined inside the loop body, a nested function which refers to a loop variable cannot see or change the current iteration's value. Instead, pass the variable explicitly or bind its value to a parameter. Up to 19 variables are supported, if supported by the enumerator. Variables can be omitted. For example, for , value in myMap calls myMap's enumerator with only its second parameter, omitting its first parameter. If the enumerator is user-defined and the parameter is mandatory, an exception is thrown as usual. The parameter count passed to __Enum is 0 if there are no variables or commas; otherwise it is 1 plus the number of commas present. Expression Type: Object An expression which results in an enumerable object, or a variable which contains an enumerable object. Remarks The parameter list can optionally be enclosed in parentheses. For example: for (val in myarray) The process of enumeration is as follows: Before the loop begins, Expression is evaluated to determine the target object. The object's __Enum method is called to retrieve an enumerator object. If no such method exists, the object itself is assumed to be an enumerator object. At the beginning of each iteration, the enumerator is called to retrieve the next value or pair of values. If it returns false (zero or an empty string), the loop terminates. Although not exactly equivalent to a for-loop, the following demonstrates this process: _enum := Expression try _enum := _enum.__Enum(2) while _enum(&Value1, &Value2) { ... } As in the code above, an exception is thrown if Expression or __Enum yields a value which cannot be called. While enumerating properties, methods or array elements, it is generally unsafe to insert or remove items of that type. Doing so may cause some items to be skipped or enumerated multiple times. One workaround is to build a list of items to remove, then use a second loop to remove the items after the first loop completes. A for-loop is usually followed by a block, which is a collection of statements that form the body of the loop. However, a loop with only a single statement does not require a block (an "if" and its "else" count as a single statement for this purpose). The One True Brace (OTB) style may optionally be used, which allows the open-brace to appear on the same line rather than underneath. For example: for x, y in z {. As with all loops, Break, Continue and A_Index may be used. The loop may optionally be followed by an Else statement, which is executed if the loop had zero iterations. COM Objects Since Value1 and Value2 are passed directly to the enumerator, the values they are assigned depends on what type of object is being enumerated. For COM objects, Value1 contains the value returned by IEnumVARIANT::Next() and Value2 contains a number which represents its variant type. For example, when used with a Scripting.Dictionary object, each Value1 contains a key from the dictionary and Value2 is typically 8 for strings and 3 for integers. See ComObjType for a list of type codes. When enumerating a SafeArray, Value1 contains the current element and Value2 contains its variant type. Related Enumerator object, OwnProps, While-loop, Loop, Until, Break, Continue, Blocks Examples Lists the properties owned by an object. colours := {red: 0xFF0000, blue: 0x0000FF, green: 0x00FF00} ; The above expression could be used directly in place of "colours" below: s := "" for k, v in colours.OwnProps() s .= k "=" v "`n" MsgBox s Lists all open Explorer and Internet Explorer windows, using the Shell object. windows := "" for window in ComObject("Shell.Application").Windows windows .= window.LocationName " :: " window.LocationURL "`n" MsgBox windows Define an enumerator as a fat arrow function. Returns numbers from the Fibonacci sequence, indefinitely or until stopped. for n in FibF() if MsgBox("#" A_Index " = " n "`nContinue?",, "y/n") = "No" break FibF() { a := 0, b := 1 return (&n) => ( n := c := b, b += a, a := c, true ) } Define an enumerator as a class. Equivalent to the previous example. for n in FibC() if MsgBox("#" A_Index " = " n "`nContinue?",, "y/n") = "No" break class FibC { a := 0, b := 1 Call(&n) { n := c := this.b, this.b += this.a, this.a := c return true } } Format - Syntax & Usage | AutoHotkey v2 Format Formats a variable number of input values according to a format string. String := Format(FormatStr , Values...) Parameters FormatStr Type: String A format string composed of literal text and placeholders of the form {Index:Format}. Index is an integer indicating which input value to use, where 1 is the first value. Format is an optional format specifier, as described below. Omit the index to use the next input value in the sequence (even if it has been used earlier in the string). For example, "{2:i} {:i}" formats the second and third input values as decimal integers, separated by a space. If Index is omitted, Format must still be preceded by :. Specify empty braces to use the next input value with default formatting: {} Use {{} and {}} to include literal braces in the string. Any other invalid placeholders are included in the result as is. Whitespace inside the braces is not permitted (except as a flag). Values Type: String, Integer or Float Input values to be formatted and inserted into the final string. Each value is a separate parameter. The first value has an index of 1. To pass an array of values, use a variadic function call: arr := [13, 240] MsgBox Format("{2:x}{1:02x}", arr*) Return Value Type: String This function returns the formatted version of the specified string. Format Specifiers Each format specifier can include the following components, in this order (without the spaces): Flags Width .Precision ULT Type Flags: Zero or more flags from the flag table below to affect output justification and prefixes. Width: A decimal integer which controls the minimum width of the formatted value, in characters. By default, values are right-aligned and spaces are used for padding. This can be overridden by using the - (left-align) and 0 (zero prefix) flags. .Precision: A decimal integer which controls the maximum number of string characters, decimal places, or significant digits to output, depending on the output type. It must be preceded by a decimal point. Specifying a precision may cause the value to be truncated or rounded. Output types and how each is affected by the precision value are as follows (see table below for an explanation of the different output types): f, e, E: Precision specifies the number of digits after the decimal point. The default is 6. g, G: Precision specifies the maximum number of significant digits. The default is 6. s: Precision specifies the maximum number of characters to be printed. Characters in excess of this are not printed. For the integer types (d, i, u, x, X, o), Precision acts like Width with the 0 prefix and a default of 1. ULT: Specifies a case transformation to apply to a string value -- Upper, Lower or Title. Valid only with the s type. For example {:U} or {:.20Ts}. Lower-case l and t are also supported, but u is reserved for unsigned integers. Type: A character from the type table below indicating how the input value should be interpreted. If omitted, it defaults to s. Flags FlagMeaning - Left align the result within the given field width (insert spaces to the right if needed). For example, Format("{:-10}", 1) returns 1 . If omitted, the result is right aligned within the given field width. + Use a sign (+ or -) to prefix the output value if it is of a signed type. For example, Format("{:+d}", 1) returns +1. If omitted, a sign appears only for negative signed values (-). 0 If width is prefixed by 0, leading zeros are added until the minimum width is reached. For example, Format("{:010}", 1) returns 0000000001. If both 0 and - appear, the 0 is ignored. If 0 is specified as an integer format (i, u, x, X, o, d) and a precision specification is also present - for example, {:04.d} - the 0 is ignored. If omitted, no padding occurs. Use a space to prefix the output value with a single space if it is signed and positive. The space is ignored if both and + flags appear. For example, Format("{: 05d}", 1) returns 0001. If omitted, no space appears. # When it's used with the o, x, or X format, the # flag uses 0, 0x, or 0X, respectively, to prefix any nonzero output value. For example, Format("{:#x}", 1) returns 0x1. When it's used with the e, E, f, a or A format, the # flag forces the output value to contain a decimal point. For example, Format("{:#.0f}", 1) returns 1.. When it's used with the g or G format, the # flag forces the output value to contain a decimal point and prevents the truncation of trailing zeros. Ignored when used with c, d, i, u, or s. Types Type CharacterArgumentOutput format d or i Integer Signed decimal integer. For example, Format("{:d}", 1.23) returns 1. u Integer Unsigned decimal integer. x or X Integer Unsigned hexadecimal integer; uses "abcdef" or "ABCDEF" depending on the case of x. The 0x prefix is not included unless the # flag is used, as in {:#x}. To always include the prefix, use 0x{:x} or similar. For example, Format("{:X}", 255) returns FF. o Integer Unsigned octal integer. For example, Format ("{:o}", 255) returns 377. f Floating-point Signed value that has the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the

decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision. For example, Format ("{:.2f}", 1) returns 1.00. e Floating-point Signed value that has the form [-]d.dddd e [sign]dd[d] where d is one decimal digit, dddd is one or more decimal digits, dd[d] is two or three decimal digits depending on the output format and size of the exponent, and sign is + or -. For example, Format("{:e}", 255) returns 2.550000e+002. E Floating-point Identical to the e format except that E rather than e introduces the exponent. g Floating-point Signed values are displayed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it. G Floating-point Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate). a Floating-point Signed hexadecimal double-precision floating-point value that has the form [?]0xh.hhhh p±dd, where h.hhhh are the hex digits (using lower case letters) of the mantissa, and dd are one or more digits for the exponent. The precision specifies the number of digits after the point. For example, Format("{:a}", 255) returns 0x1.fe0000p+7. A Floating-point Identical to the a format, except that P, rather than p, introduces the exponent. p Integer Displays the argument as a memory address in hexadecimal digits. For example, Format("{:p}", 255) returns 000000FF. s String Specifies a string. If the input value is numeric, it is automatically converted to a string before the Width and Precision arguments are applied. c Character code Specifies a single character by its ordinal value, similar to Chr(n). If the input value is outside the expected range, it wraps around. For example, Format("{:c}", 116) returns t. Remarks Unlike printf, size specifiers are not supported. All integers and floating-point input values are 64-bit. Related FormatTime Examples Demonstrates different usages. s := "" ; Simple substitution s .= Format("{2}, {1}!`r`n", "World", "Hello") ; Padding with spaces s .= Format("|{:-10}|`r`n|{:10}|`r`n", "Left", "Right") ; Hexadecimal s .= Format("{1:#x} {2:X} 0x{3:x}`r`n", 3735928559, 195948557, 0) ; Floating-point s .= Format ("{1:0.3f} {1:.10f}", 4*ATan(1)) ListVars ; Use AutoHotkey's main window to display monospaced text. WinWaitActive "ahk_class AutoHotkey" ControlSetText(s, "Edit1") WinWaitClose FormatTime - Syntax & Usage | AutoHotkey v2 FormatTime Transforms a YYYYMMDDHH24MISS timestamp into the specified date/time format. String := FormatTime(YYYYMMDDHH24MISS, Format) Parameters YYYYMMDD... Type: String Leave this parameter blank to use the current local date and time. Otherwise, specify all or the leading part of a timestamp in the YYYYMMDDHH24MISS format. Format Type: String If omitted, it defaults to the time followed by the long date, both of which will be formatted according to the current user's locale. For example: 4:55 PM Saturday, November 27, 2004 Otherwise, specify one or more of the date-time formats from the tables below, along with any literal spaces and punctuation in between (commas do not need to be escaped; they can be used normally). In the following example, note that M must be capitalized: M/d/yyyy h:mm tt Return Value Type: String This function returns the transformed version of the specified timestamp. If YYYYMMDDHH24MISS contains a invalid date and/or time portion -- such as February 29th of a non-leap year -- the date and/or time will be omitted from the return value. Although only years between 1601 and 9999 are supported, a formatted time can still be produced for earlier years as long as the time portion is valid. If Format contains more than 2000 characters, an empty string is returned. Date Formats (case sensitive) Format Description d Day of the month without leading zero (1 – 31) dd Day of the month with leading zero (01 – 31) ddd Abbreviated name for the day of the week (e.g. Mon) in the current user's language dddd Full name for the day of the week (e.g. Monday) in the current user's language M Month without leading zero (1 – 12) MM Month with leading zero (01 – 12) MMM Abbreviated month name (e.g. Jan) in the current user's language MMMM Full month name (e.g. January) in the current user's language y Year without century, without leading zero (0 – 99) yy Year without century, with leading zero (00 – 99) yyyy Year with century. For example: 2005 gg Period/era string for the current user's locale (blank if none) Time Formats (case sensitive) Format Description h Hours without leading zero; 12-hour format (1 – 12) hh Hours with leading zero; 12-hour format (01 – 12) H Hours without leading zero; 24-hour format (0 – 23) HH Hours with leading zero; 24-hour format (00 – 23) m Minutes without leading zero (0 – 59) mm Minutes with leading zero (00 – 59) s Seconds without leading zero (0 – 59) ss Seconds with leading zero (00 – 59) t Single character time marker, such as A or P (depends on locale) tt Multi-character time marker, such as AM or PM (depends on locale) Standalone Formats The following formats must be used alone; that is, with no other formats or text present in the Format parameter. These formats are not case sensitive. Format Description (Blank) Leave Format blank to produce the time followed by the long date. For example, in some locales it might appear as 4:55 PM Saturday, November 27, 2004 Time Time representation for the current user's locale, such as 5:26 PM ShortDate Short date representation for the current user's locale, such as 02/29/04 LongDate Long date representation for the current user's locale, such as Friday, April 23, 2004 YearMonth Year and month format for the current user's locale, such as February, 2004 YDay Day of the year without leading zeros (1 – 366) YDay0 Day of the year with leading zeros (001 – 366) WDay Day of the week (1 – 7). Sunday is 1. YWeek The ISO 8601 full year and week number. For example: 200453. 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. Consequently, both January 4th and the first Thursday of January are always in week 1. Additional Options The following options can appear inside the YYYYMMDDHH24MISS parameter immediately after the timestamp (if there is no timestamp, they may be used alone). In the following example, note the lack of commas between the last four items: OutputVar := FormatTime("20040228 LSys D1 D4") R: Reverse. Have the date come before the time (meaningful only when Format is blank). Ln: If this option is not present, the current user's locale is used to format the string. To use the system's locale instead, specify LSys. To use a specific locale, specify the letter L followed by a hexadecimal or decimal locale identifier (LCID). For information on how to construct an LCID, search www.microsoft.com for the following phrase: Locale Identifiers Dn: Date options. Specify for n one of the following numbers: 0 = Force the default options to be used. This also causes the short date to be in effect. 1 = Use short date (meaningful only when Format is blank; not compatible with 2 and 8). 2 = Use long date (meaningful only when Format is blank; not compatible with 1 and 8). 4 = Use alternate calendar (if any). 8 = Use Year-Month format (meaningful only when Format is blank; not compatible with 1 and 2). 0x10 = Add marks for left-to-right reading order layout. 0x20 = Add marks for right-to-left reading order layout. 0x80000000 = Do not obey any overrides the user may have in effect for the system's default date format. 0x40000000 = Use the system ANSI code page for string translation instead of the locale's code page. Tn: Time options. Specify for n one of the following numbers: 0 = Force the default options to be used. This also causes minutes and seconds to be shown. 1 = Omit minutes and seconds. 2 = Omit seconds. 4 = Omit time marker (e.g. AM/PM). 8 = Always use 24-hour time rather than 12-hour time. 12 = Combination of the above two. 0x80000000 = Do not obey any overrides the user may have in effect for the system's default time format. 0x40000000 = Use the system ANSI code page for string translation instead of the locale's code page. Note: Dn and Tn may be repeated to put more than one option into effect, such as this example: FormatTime ("20040228 D2 D4 T1 T8") Remarks Letters and numbers that you want to be transcribed literally from Format into the final string should be enclosed in single quotes as in this example: "'Date:' MM/dd/yy 'Time:' hh:mm:ss tt". By contrast, non-alphanumeric characters such as spaces, tabs, linefeeds (`n), slashes, colons, commas, and other punctuation do not need to be enclosed in single quotes. The exception to this is the single quote character itself: to produce it literally, use four consecutive single quotes (""), or just two if the quote is already inside an outer pair of quotes. If Format contains date and time elements together, they must not be intermixed. In other words, the string should be dividable into two halves: a time half and a date half. For example, a format string consisting of "hh yyyy mm" would not produce the expected result because it has a date element in between two time elements. When Format contains a numeric day of the month (either d or dd) followed by the full month name (MMMM), the genitive form of the month name is used (if the language has a genitive form). On a related note, addition, subtraction and comparison of dates and times can be performed with DateAdd and DateDiff. Related To convert in the reverse direction -- that is, from a formatted date/time to YYYYMMDDHH24MISS format -- see www.autohotkey.com/forum/topic20405.html See also: Gui DateTime control, Format, built-in date and time variables, FileGetTime Examples Demonstrates different usages. TimeString := FormatTime() MsgBox "The current time and date (time first) is " TimeString TimeString := FormatTime("R") MsgBox "The current time and date (date first) is " TimeString TimeString := FormatTime(, "Time") MsgBox "The current time is " TimeString TimeString := FormatTime("T12", "Time") MsgBox "The current 24-hour time is " TimeString TimeString := FormatTime(, "LongDate") MsgBox "The current date (long format) is " TimeString TimeString := FormatTime(20050423220133, "dddd MMMM d, yyyy hh:mm:ss tt") MsgBox "The specified date and time, when formatted, is " TimeString MsgBox FormatTime(200504, "'Month Name': MMMM`n'Day Name': dddd") YearWeek := FormatTime(20050101, "YWeek") MsgBox "January 1st of 2005 is in the following ISO year and week number: " YearWeek Changes the date-time stamp of a file. FileName := FileSelect(3,, "Pick a file") if FileName = "" ; The user didn't pick a file. return FileTime := FileGetTime(FileName) FileTime := FormatTime(FileTime) ; Since the last parameter is omitted, the long date and time are retrieved. MsgBox "The selected file was last modified at " FileTime Converts the specified number of seconds into the corresponding number of hours, minutes, and seconds (hh:mm:ss format). MsgBox FormatSeconds(7384) ; 7384 = 2 hours + 3 minutes + 4 seconds. It yields: 2:03:04 FormatSeconds(NumberOfSeconds) ; Convert the specified number of seconds to hh:mm:ss format. { time := 19990101 ; *Midnight* of an arbitrary date. time := DateAdd(time, NumberOfSeconds, "Seconds") return NumberOfSeconds//3600 ":" FormatTime(time, "mm:ss") /* ; Unlike the method used above, this would not support more than 24 hours worth of seconds: return FormatTime(time, "h:mm:ss") */ } Func Object - Methods & Properties | AutoHotkey v2 Func Object class Func extends Object Represents a user-defined or built-in function. For information about other objects which can be called like functions, see Function Objects. The Closure class extends Func but does not define any new properties. For each built-in function or function definition within the script, there is a corresponding read-only variable containing a Func object. This variable is directly used to call the function, but its value can also be read to retrieve the function itself, as a value. For example: InspectFn StrLen InspectFn InspectFn InspectFn(fn) { ; Display information about the passed function. MsgBox fn.Name "() is " (fn.IsBuiltIn ? "built-in." : "user-defined.") } "FuncObj" is used below as a placeholder for any Func object, as "Func" is the class itself. In addition to the methods and property inherited from Object, Func objects have the following predefined methods and properties. Table of Contents Methods: Call: Calls the function. Bind: Binds parameters to the function and returns a BoundFunc object. IsByRef: Determines whether a parameter is ByRef. IsOptional: Determines whether a parameter is optional. Properties: Name: Returns the function's name. IsBuiltIn: Returns true if the function is built-in and false otherwise. IsVariadic: Returns true if the function is variadic and false otherwise. MinParams: Returns the number of required parameters. MaxParams: Returns the number of formally-declared parameters for a user-defined function or maximum parameters for a built-in function. Methods Call Calls the function.

FuncObj(Param1, Param2, ...) FuncObj.Call(Param1, Param2, ...) Param1, Param2, ... Parameters and return value are defined by the function. The "Call" method is implied when calling a value, so need not be explicitly specified. Bind Binds parameters to the function and returns a BoundFunc object. BoundFunc := FuncObj.Bind(Param1, Param2, ...) Param1, Param2, ... Any number of parameters. For details and examples, see BoundFunc object. IsByRef Determines whether a parameter is ByRef. Boolean := FuncObj.IsByRef(ParamIndex) ParamIndex Type: Integer Optional: the one-based index of a parameter. If omitted, Boolean indicates whether the function has any ByRef parameters. Returns a boolean value indicating whether the parameter is ByRef. If ParamIndex is invalid, an exception is thrown. IsOptional Determines whether a parameter is optional. Boolean := FuncObj.IsOptional(ParamIndex) ParamIndex Type: Integer Optional: the one-based index of a parameter. If omitted, Boolean indicates whether the function has any optional parameters. Returns a boolean value indicating whether the parameter is optional. If ParamIndex is invalid, an exception is thrown. Parameters do not need to be formally declared if the function is variadic. Built-in functions are supported. Properties Name Returns the function's name. FunctionName := FuncObj.Name IsBuiltIn Returns true if the function is built-in and false otherwise. Boolean := FuncObj.IsBuiltIn IsVariadic Returns true if the function is variadic and false otherwise. Boolean := FuncObj.IsVariadic MinParams Returns the number of required parameters. ParamCount := FuncObj.MinParams MaxParams Returns the number of formally-declared parameters for a user-defined function or maximum parameters for a built-in function. ParamCount := FuncObj.MaxParams If the function is variadic, ParamCount indicates the maximum number of parameters which can be accepted by the function without overflowing into the "variadic*" parameter. GetKeyName - Syntax & Usage | AutoHotkey v2 GetKeyName Retrieves the name or text of a key. Name := GetKeyName(KeyName) Parameters KeyName Type: String This can be just about any single character from the keyboard or one of the key names from the key list. Examples: B, 5, LWin, RControl, Alt, Enter, Escape. Alternatively, this can be an explicit virtual key code such as vkFF, an explicit scan code such as sc01D, or a combination of VK and SC (in that order) such as vk1Bsc001. Note that these codes must be in hexadecimal. Return Value Type: String This function returns the name of the specified key, or blank if the key is invalid or unnamed. Related GetKeyVK, GetKeySC, GetKeyState, Key List Examples Retrieves and reports the English name of Esc. MsgBox GetKeyName("Esc") ; Shows Escape MsgBox GetKeyName ("vk1B") ; Shows also Escape GetKeySC - Syntax & Usage | AutoHotkey v2 GetKeySC Retrieves the scan code of a key. SC := GetKeySC(KeyName) Parameters KeyName Type: String Any single character or one of the key names from the key list. Examples: B, 5, LWin, RControl, Alt, Enter, Escape. Alternatively, this can be an explicit virtual key code such as vkFF, an explicit scan code such as sc01D, or a combination of VK and SC (in that order) such as vk1Bsc001. Note that these codes must be in hexadecimal. Return Value Type: Integer This function returns the scan code of the specified key, or 0 if the key is invalid or has no scan code. Remarks Before using the scan code with a built-in function like Hotkey or GetKeyState, it must first be converted to hexadecimal format, such as by using Format ("sc{:X}", sc_code). By contrast, external functions called via DllCall typically use the numeric value directly. If KeyName corresponds to a virtual key code or single character, the function attempts to map the value to a scan code by calling certain system functions which refer to the script's current keyboard layout. This may differ from the keyboard layout of the active window. If KeyName is an ASCII letter in the range A-Z and has no mapping within the keyboard layout, the corresponding virtual key in the range vk41-vk5A is used as a fallback. This virtual key is then mapped to a scan code as described above. Some keyboard layouts do not define a 1:1 mapping of virtual key codes to scan codes. When multiple interpretations are possible, the underlying system functions most likely choose one based on the order defined in the keyboard layout, which is not always the most common or logical choice. Related GetKeyVK, GetKeyName, GetKeyState, Key List, Format Examples Retrieves and reports the hexadecimal scan code of the left Ctrl. sc_code := GetKeySC("LControl") MsgBox Format("sc{:X}", sc_code) ; Reports sc1D GetKeyState - Syntax & Usage | AutoHotkey v2 GetKeyState Checks if a keyboard key or mouse/joystick button is down or up. Also retrieves joystick status. IsDown := GetKeyState(KeyName , Mode) Parameters KeyName Type: String This can be just about any single character from the keyboard or one of the key names from the key list, such as a mouse/joystick button. Examples: B, 5, LWin, RControl, Alt, Enter, Escape, LButton, MButton, Joy1. Alternatively, an explicit virtual key code such as vkFF may be specified. This is useful in the rare case where a key has no name. The code of such a key can be determined by following the steps at the bottom of the key list page. Note that this code must be in hexadecimal. Known limitation: This function cannot differentiate between two keys which share the same virtual key code, such as Left and NumpadLeft. Mode Type: String This parameter is ignored when retrieving joystick status. If omitted, the mode will default to that which retrieves the logical state of the key. This is the state that the OS and the active window believe the key to be in, but is not necessarily the same as the physical state. Alternatively, one of these letters may be specified: P: Retrieve the physical state (i.e. whether the user is physically holding it down). The physical state of a key or mouse button will usually be the same as the logical state unless the keyboard and/or mouse hooks are installed, in which case it will accurately reflect whether or not the user is physically holding down the key or button (as long as it was pressed down while the script was running). You can determine if your script is using the hooks via the KeyHistory function or menu item. You can force the hooks to be installed by calling InstallKeybdHook and/or InstallMouseHook. T: Retrieve the toggle state. For keys other than CapsLock, NumLock and ScrollLock, the toggle state is generally 0 when the script starts and is not synchronized between processes. Return Value Type: Integer (boolean), Float, Integer or String (empty) This function returns 1 (true) if the key is down (or toggled on) or 0 (false) if it is up (or toggled off). When KeyName is a joystick axis such as JoyX, this function returns a floating-point number between 0 and 100 to indicate the joystick's position as a percentage of that axis's range of motion. This test script can be used to analyze your joystick(s). When KeyName is JoyPOV, this function returns an integer between 0 and 35900. The following approximate POV values are used by many joysticks: -1: no angle to report 0: forward POV 9000 (i.e. 90 degrees): right POV 27000 (i.e. 270 degrees): left POV 18000 (i.e. 180 degrees): backward POV When KeyName is a button or control of a joystick that could not be detected, this function returns an empty string. Error Handling A ValueError is thrown if invalid parameters are detected, e.g. when KeyName does not exist on the current keyboard layout. Remarks To wait for a key or mouse/joystick button to achieve a new state, it is usually easier to use KeyWait instead of a GetKeyState loop. Systems with unusual keyboard drivers might be slow to update the state of their keys, especially the toggle-state of keys like CapsLock. A script that checks the state of such a key immediately after it changed may use Sleep beforehand to give the system time to update the key state. For examples of using GetKeyState with a joystick, see the joystick remapping page and the Joystick-To-Mouse script. Related GetKeyVK, GetKeySC, GetKeyName, KeyWait, Key List, Joystick remapping, KeyHistory, InstallKeybdHook, InstallMouseHook Examples Retrieves the current state of the right mouse button. state := GetKeyState("RButton") Retrieves the current state of the first joystick's second button. state := GetKeyState("Joy2") Checks if at least one Shift is down. if GetKeyState("Shift") MsgBox "At least one Shift key is down." else MsgBox "Neither Shift key is down." Retrieves the current toggle state of CapsLock. state := GetKeyState("CapsLock", "T") Remapping. (This example is only for illustration because it would be easier to use the built-in remapping feature.) In the following hotkey, the mouse button is kept held down while NumpadAdd is down, which effectively transforms NumpadAdd into a mouse button. This method can also be used to repeat an action while the user is holding down a key or button. *NumpadAdd:: { MouseClick "left",,, 1, 0, "D" ; Hold down the left mouse button. Loop { Sleep 10 if !GetKeyState("NumpadAdd", "P") ; The key has been released, so break out of the loop. break ; ... insert here any other actions you want repeated. } MouseClick "left",,, 1, 0, "U" ; Release the mouse button. } Makes joystick button behavior depend on joystick axis position. joy2:: { JoyX := GetKeyState("JoyX") if JoyX > 75 MsgBox "Action #1 (button pressed while joystick was pushed to the right)." else if JoyX < 25 MsgBox "Action #2 (button pressed while joystick was pushed to the left)." else MsgBox "Action #3 (button pressed while joystick was centered horizontally)." } See the joystick remapping page and the Joystick-To-Mouse script for other examples. GetKeyVK - Syntax & Usage | AutoHotkey v2 GetKeyVK Retrieves the virtual key code of a key. VK := GetKeyVK(KeyName) Parameters KeyName Type: String Any single character or one of the key names from the key list. Examples: B, 5, LWin, RControl, Alt, Enter, Escape. Alternatively, this can be an explicit virtual key code such as vkFF, an explicit scan code such as sc01D, or a combination of VK and SC (in that order) such as vk1Bsc001. Note that these codes must be in hexadecimal. Return Value Type: Integer This function returns the virtual key code of the specified key, or 0 if the key is invalid or has no virtual key code. Remarks Before using the virtual key code with a built-in function like Hotkey or GetKeyState, it must first be converted to hexadecimal format, such as by using Format("vk{:X}", vk_code). By contrast, external functions called via DllCall typically use the numeric value directly. If KeyName corresponds to a scan code or single character, the function attempts to map the value to a virtual key code by calling certain system functions which refer to the script's current keyboard layout. This may differ from the keyboard layout of the active window. If KeyName is an ASCII letter in the range A-Z and has no mapping within the keyboard layout, the corresponding virtual key in the range vk41-vk5A is used as a fallback. Some keyboard layouts do not define a 1:1 mapping of virtual key codes to scan codes. When multiple interpretations are possible, the underlying system functions most likely choose one based on the order defined in the keyboard layout, which is not always the most common or logical choice. Related GetKeySC, GetKeyName, GetKeyState, Key List, Format Examples Retrieves and reports the hexadecimal virtual key code of Esc. vk_code := GetKeyVK("Esc") MsgBox Format("vk{:X}", vk_code) ; Reports vk1B GetMethod - Syntax & Usage | AutoHotkey v2 GetMethod Retrieves the implementation function of a method. Method := GetMethod(Value , Name, ParamCount) Parameters Value Type: Any Any value, of any type except ComObject. Name Type: String The name of the method to retrieve. Omit this parameter to perform validation on Value itself and return Value if successful. ParamCount Type: Integer The number of parameters that would be passed to the method or function. If specified, the method's MinParams, MaxParams and IsVariadic properties may be queried to verify that it can accept this number of parameters. If those properties are not present, the parameter count is not verified. This count should not include the implicit this parameter. If omitted (or if the parameter count was not verified), a basic check is performed for a Call method to verify that the object is most likely callable. Return Value Type: Function Object This function returns the function object which contains the implementation of the method, or Value itself if Name was omitted. Errors If the method is not found or cannot be retrieved without invoking a property getter, a MethodError is thrown. If validation is attempted, exceptions may be thrown as a result of querying the method's properties. A ValueError or MethodError is thrown if validation fails. Remarks Methods may be defined through one of the following: A dynamic property with a call accessor function. This includes: Any property created by a method definition within a class. Any property created by passing a descriptor like {Call: fn} to DefineProp, where fn implements the method. Any predefined/built-in method. An own value property of the object or one of its base objects, where the value is a

function object. A dynamic property with a getter which returns a function object. This case is not supported by GetMethod. Handling within a __Call meta-function. Methods implemented this way cannot be detected and may not even have a corresponding function object, so are not supported by GetMethod. When calling the function object, it is necessary to supply a value for the normally-hidden this parameter. For example, Method(Value, Parameters*). Although the standard implementation of GetMethod has limitations as described above, if Value.GetMethod(Name) is used instead of GetMethod(Value, Name), the object Value can define its own implementation of GetMethod. GetMethod(Value, "Call", N) is not the same as GetMethod(Value,, N), as the Call method takes the function object itself as a parameter, and its usage may otherwise differ from that of Value. For instance, Func.Prototype.Call is a single method which applies to all built-in and user-defined functions, and as such must accept any number of parameters. Related Objects, HasMethod, HasBase, HasProp Examples Retrieves and reports information about the GetMethod method. method := GetMethod({}, "GetMethod") ; It's also a method. MsgBox method.MaxParams ; Takes 2 parameters, including 'this'. MsgBox method = GetMethod ; Actually the same object in this case. Goto - Syntax & Usage | AutoHotkey v2 Goto Jumps to the specified label and continues execution. Goto Label Goto("Label") Parameters Label Type: String The name of the label to which to jump. Remarks Label can be a variable or expression only if parentheses are used. For example, Goto MyLabel and Goto("MyLabel") both jump to MyLabel:. Performance is slightly reduced when using a dynamic label (that is, a variable or expression which returns a label name) because the target label must be "looked up" each time rather than only once when the script is first loaded. An error dialog will be displayed if the label does not exist. To avoid this, call IsLabel() beforehand. For example: if IsLabel (VarContainingLabelName) Goto(VarContainingLabelName) The use of Goto is discouraged because it generally makes scripts less readable and harder to maintain. Consider using Else, Blocks, Break, and Continue as substitutes for Goto. Related Return, IsLabel, Else, Blocks, Break, Continue Examples Jumps to the label named "MyLabel" and continues execution. Goto MyLabel ; ... MyLabel: Sleep 100 ; ... GroupActivate - Syntax & Usage | AutoHotkey v2 GroupActivate Activates the next window in a window group that was defined with GroupAdd. HWND := GroupActivate(GroupName , Mode) Parameters GroupName Type: String The name of the group to activate, as originally defined by GroupAdd. Mode Type: String If omitted, the function activates the oldest window in the series. To change this behavior, specify the following letter: R: The newest window (the one most recently active) is activated, but only if no members of the group are active when the function is given. "R" is useful in cases where you temporarily switch to working on an unrelated task. When you return to the group via GroupActivate, GroupDeactivate, or GroupClose, the window you were most recently working with is activated rather than the oldest window. Return Value Type: Integer This function returns the HWND (unique ID) of the window selected for activation, or 0 if no matching windows were found to activate. If the current active window is the only match, the return value is 0. Remarks This function causes the first window that matches one of the group's window specifications to be activated. Using it a second time will activate the next window in the series and so on. Normally, it is assigned to a hotkey so that this window-traversal behavior is automated by pressing that key. Each window is evaluated against the window group as a whole, without distinguishing between window specifications. Mode affects the order of activation across the entire group. When a window is activated immediately after another window was activated, task bar buttons may start flashing on some systems (depending on OS and settings). To prevent this, use #WinActivateForce. See GroupAdd for more details about window groups. Related GroupAdd, GroupDeactivate, GroupClose, #WinActivateForce Examples Activates the newest window (the one most recently active) in a window group. GroupActivate "MyGroup", "R" GroupAdd - Syntax & Usage | AutoHotkey v2 GroupAdd Adds a window specification to a window group, creating the group if necessary. GroupAdd GroupName , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters GroupName Type: String The name of the group to which to add this window specification. If the group doesn't exist, it will be created. Group names are not case sensitive. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON at the time that GroupActivate, GroupDeactivate, and GroupClose are used. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Remarks Each use of this function adds a new rule to a group. In other words, a group consists of a set of criteria rather than a fixed list of windows. Later, when a group is used by a function such as GroupActivate, each window on the desktop is checked against each of these criteria. If a window matches one of the criteria in the group, it is considered a match. Although SetTitleMatchMode and DetectHiddenWindows do not directly affect the behavior of this function, they do affect the other group functions such as GroupActivate and GroupClose. They also affect the use of ahk_group in any other function's WinTitle. A window group is typically used to bind together a collection of related windows, which is useful for tasks that involve many related windows, or an application that owns many subwindows. For example, if you frequently work with many instances of a graphics program or text editor, you can use GroupActivate on a hotkey to visit each instance of that program, one at a time, without having to use alt-tab or task bar buttons to locate them. Since the entries in each group need to be added only once, this function is typically used during script startup. Attempts to add duplicate entries to a group are ignored. To include all windows in a group (except the special Program Manager window), use this example: GroupAdd "AllWindows" All windowing functions can operate upon a window group by specifying ahk_group MyGroupName for the WinTitle parameter. The functions WinMinimize, WinMaximize, WinRestore, WinHide, WinShow, WinClose, and WinKill will act upon all the group's windows. To instead act upon only the topmost window, follow this example: WinHide WinExist("ahk_group MyGroup") By contrast, the other window functions such as WinActivate and WinExist will operate only upon the topmost window of the group. Related GroupActivate, GroupDeactivate, GroupClose Examples Press a hotkey to traverse all open MSIE windows. ; In global code, to be evaluated at startup: GroupAdd "MSIE", "ahk_class IEFrame" ; Add only Internet Explorer windows to this group. ; Assign a hotkey to activate this group, which traverses ; through all open MSIE windows, one at a time (i.e. each ; press of the hotkey). Numpad1::GroupActivate "MSIE", "r" Press a hotkey to visit each MS Outlook 2002 window, one at a time. ; In global code, to be evaluated at startup: SetTitleMatchMode 2 GroupAdd "mail", "Message - Microsoft Word" ; This is for mails currently being composed GroupAdd "mail", "- Message (" ; This is for already opened items ; Need extra text to avoid activation of a phantom window: GroupAdd "mail", "Advanced Find", "Sear&ch for the word(s)" GroupAdd "mail", , "Recurrence:" GroupAdd "mail", "Reminder" GroupAdd "mail", "- Microsoft Outlook" ; Assign a hotkey to visit each Outlook window, one at a time. Numpad5::GroupActivate "mail" GroupClose - Syntax & Usage | AutoHotkey v2 GroupClose Closes the active window if it was just activated by GroupActivate or GroupDeactivate. It then activates the next window in the series. It can also close all windows in a group. GroupClose GroupName , Mode Parameters GroupName Type: String The name of the group as originally defined by GroupAdd. Mode Type: String If omitted, the function closes the active window and activates the oldest window in the series. To change this behavior, specify one of the following letters: R: The newest window (the one most recently active) is activated, but only if no members of the group are active when the function is given. "R" is useful in cases where you temporarily switch to working on an unrelated task. When you return to the group via GroupActivate, GroupDeactivate, or GroupClose, the window you were most recently working with is activated rather than the oldest window. A: All members of the group will be closed. This is the same effect as WinClose "ahk_group GroupName". Remarks When the Mode parameter is not "A", the behavior of this function is determined by whether the previous action on GroupName was GroupActivate or GroupDeactivate. If it was GroupDeactivate, this function will close the active window only if it is not a member of the group (otherwise it will do nothing). If it was GroupActivate or nothing, this function will close the active window only if it is a member of the group (otherwise it will do nothing). This behavior allows GroupClose to be assigned to a hotkey as a companion to GroupName's GroupActivate or GroupDeactivate hotkey. When the active window closes, the system typically activates the next most recently active window. If the newly active window is a match for the same window specification as the window that was just closed, it is left active even though the default Mode would normally dictate that the oldest window should be activated next. If the newly active window is a match for any of the group's window specifications, it is left active. See GroupAdd for more details about window groups. Related GroupAdd, GroupActivate, GroupDeactivate Examples Closes the active window activated by GroupActivate or GroupDeactivate and activates the newest window (the one most recently active) in a window group. GroupClose "MyGroup", "R" GroupDeactivate - Syntax & Usage | AutoHotkey v2 GroupDeactivate Similar to GroupActivate except activates the next window not in the group. GroupDeactivate GroupName , Mode Parameters GroupName Type: String The name of the target group, as originally defined by GroupAdd. Mode Type: String If omitted, the function activates the oldest non-member window. To change this behavior, specify the following letter: R: The newest non-member window (the one most recently active) is activated, but only if a member of the group is active when the function is given. "R" is useful in cases where you temporarily switch to working on an unrelated task. When you return to the group via GroupActivate, GroupDeactivate, or GroupClose, the window you were most recently working with is activated rather than the oldest window. Remarks GroupDeactivate causes the first window that does not match any of the group's window specifications to be activated. Using GroupDeactivate a second time will activate the next window in the series and so on. Normally, GroupDeactivate is assigned to a hotkey so that this window-traversal behavior is automated by pressing that key. This function is useful in cases where you have a collection of favorite windows that are almost always running. By adding these windows to a group, you can use GroupDeactivate to visit each window that isn't one of your favorites and decide whether to close it. This allows you to clean up your desktop much more quickly than doing it manually. GroupDeactivate selects windows in a manner similar to the Alt+Shift+Esc system hotkey. Specifically: Owned windows, such as certain dialogs and tool windows, are not evaluated. However, if the owner window is eligible for activation, the most recently active owned window is activated instead, unless the owner was active more recently. This is determined by calling GetLastActivePopup. Windows with the WS_EX_TOPMOST or WS_EX_NOACTIVATE styles are skipped. Windows with the WS_EX_TOOLWINDOW style but without the WS_EX_APPWINDOW style are skipped. (These windows are generally also omitted from Alt-Tab and the taskbar.) Disabled windows are skipped, unless a window it owns (such as a modal dialog) was active more recently than the window itself. Hidden or cloaked windows are skipped. The Desktop is skipped. Although the taskbar is skipped due to the WS_EX_TOPMOST style, it is activated if there are no other eligible windows and the active window matches the group. See GroupAdd for more details about window groups. Related GroupAdd, GroupActivate, GroupClose Examples Activates the oldest window which is not a member of a window group. GroupDeactivate "MyFavoriteWindows" ; Visit non-favorite

windows to clean up desktop. Gui Object - Methods & Properties | AutoHotkey v2 Gui Object class Gui extends Object Provides an interface for creating and managing windows, and creating controls. Such windows can be used as data entry forms or custom user interfaces. Gui objects can be created with Gui() and retrieved with GuiFromHwnd. "MyGui" is used below as a placeholder for any Gui object (and a variable name in examples), as "Gui" is the class itself. In addition to the methods and property inherited from Object, Gui objects have the following predefined methods and properties. Table of Contents Static Methods: Call: Creates a new Gui object. Methods: Add: Creates a control such as text, button, or checkbox. Destroy: Deletes the window. Flash: Blinks the window and its taskbar button. GetClientPos: Retrieves the position and size of the window's client area. GetPos: Retrieves the position and size of the window. Hide: Hides the window. Maximize: Unhides and maximizes the window. Minimize: Unhides and minimizes the window. Move: Moves and/or resizes the GUI window. OnEvent: Registers a function or method to be called when the given event is raised. Opt: Sets various options and styles for the appearance and behavior of the window. Restore: Unhides and restores the window, if it was minimized or maximized beforehand. SetFont: Sets the typeface, size, style, and text color for subsequently created controls. Show: Displays the window. It can also minimize, maximize, or move the window. Submit: Collects the values from named controls and composes them into an Object. Optionally hides the window. __Enum: Allows to iterate through the GUI's controls. __New: Constructs a new Gui instance. Properties: BackColor: Retrieves or sets the background color of the window. FocusedCtrl: Retrieves the GuiControl object of the GUI's focused control. Hwnd: Retrieves the window handle (HWND) of the GUI window. MarginX: Retrieves or sets the size of horizontal margins between sides and subsequently created controls. MarginY: Retrieves or sets the size of vertical margins between sides and subsequently created controls. MenuBar: Retrieves or sets the window's menu bar. Name: Retrieves or sets a custom name for the GUI window. Title: Retrieves or sets the GUI's title. __Item: Retrieves the GuiControl object associated with the specified name, text, ClassNN or HWND. General: Keyboard Navigation Window Appearance General Remarks Related Examples Static Methods Call Creates and returns a new Gui object. MyGui := Gui(Options, Title, EventObj) Options Type: String This parameter can contain any of the options supported by Gui.Opt. Title Type: String The window title. If omitted, it defaults to the current value of A_ScriptName. EventObj Type: Object An "event sink", or object to bind events to. If EventObj is specified, OnEvent, OnNotify and OnCommand can be used to register methods of EventObj to be called when an event is raised. If omitted or empty, any string passed to OnEvent's Function parameter is interpreted as a function name. Methods Add Adds a control to the GUI window, and returns a GuiControl object. MyGui.Add(ControlType , Options, Text) MyGui.AddControlType(Options, Text) ControlType Type: String This is one of the following: Text, Edit, UpDown, Picture (or Pic), Button, Checkbox, Radio, DropDownList (or DDL), ComboBox, ListBox, ListView, TreeView, Link, Hotkey, DateTime, MonthCal, Slider, Progress, GroupBox, Tab, Tab2, Tab3, StatusBar, ActiveX, Custom For example: MyGui := Gui() MyGui.Add("Text",, "Please enter your name:") MyGui.AddEdit("vName") MyGui.Show Options Type: String Positioning and Sizing of Controls If some dimensions and/or coordinates are omitted from Options, the control will be positioned relative to the previous control and/or sized automatically according to its nature and contents. The following options are supported: R: Rows of text (can contain a floating point number such as R2.5). R is often preferable to specifying H (Height). If both the R and H options are present, R will take precedence. For a GroupBox, this setting is the number of controls for which to reserve space inside the box. For DropDownLists, ComboBoxes, and ListBoxes, it is the number of items visible at one time inside the list portion of the control (but it is often desirable to omit both the R and H options for DropDownList and ComboBox, as the popup list will automatically take advantage of the available height of the user's desktop). For other control types, R is the number of rows of text that can visibly fit inside the control. W: Width, in pixels. If omitted, the width is calculated automatically for some control types based on their contents; tab controls default to 30 times the current font size, plus 3 times the X-margin; vertical Progress Bars default to two times the current font size; and horizontal Progress Bars, horizontal Sliders, DropDownLists, ComboBoxes, ListBoxes, GroupBoxes, Edits, and Hotkeys default to 15 times the current font size (except GroupBoxes, which multiply by 18 to provide room inside for margins). H: Height, in pixels. If both the H and R options are absent, DropDownLists, ComboBoxes, ListBoxes, and empty multi-line Edit controls default to 3 rows; GroupBoxes default to 2 rows; vertical Sliders and Progress Bars default to 5 rows; horizontal Sliders default to 30 pixels (except if a thickness has been specified); horizontal Progress Bars default to 2 times the current font size; Hotkey controls default to 1 row; and Tab controls default to 10 rows. For the other control types, the height is calculated automatically based on their contents. Note that for DropDownLists and ComboBoxes, H is the combined height of the control's always-visible portion and its list portion (but even if the height is set too low, at least one item will always be visible in the list). Also, for all types of controls, specifying the number of rows via the R option is usually preferable to using H because it prevents a control from showing partial/incomplete rows of text. wp+n, hp+n, wp-n, hp-n (where n is any number) can be used to set the width and/or height of a control equal to the previously added control's width or height, with an optional plus or minus adjustment. For example, wp would set a control's width to that of the previous control, and wp-50 would set it equal to 50 less than that of the previous control. X, Y: X-position, Y-position. For example, specifying x0 y0 would position the control in the upper left corner of the window's client area, which is the area beneath the title bar and menu bar (if any). x+n, y+n (where n is any number): An optional plus sign can be included to position a control relative to the right or bottom edge (respectively) of the control that was previously added. For example, specifying Y+10 would position the control 10 pixels beneath the bottom of the previous control rather than using the standard padding distance. Similarly, specifying X+10 would position the control 10 pixels to the right of the previous control's right edge. Since negative numbers such as X-10 are reserved for absolute positioning, to use a negative offset, include a plus sign in front of it. For example: X+-10. For X+ and Y+, the letter M can be used as a substitute for the window's current margin. For example, x+m uses the right edge of the previous control plus the standard padding distance. xp y+m positions a control below the previous control, whereas specifying a relative X coordinate on its own (with xp or x+) would normally imply yp by default. xp+n, yp+n, xp-n, yp-n (where n is any number) can be used to position controls relative to the previous control's upper left corner, which is often useful for enclosing controls in a GroupBox. xm and ym can be used to position a control at the leftmost and topmost margins of the window, respectively (these two may also be followed by a plus/minus sign and a number). xs and ys: these are similar to xm and ym except that they refer to coordinates that were saved by having previously added a control with the word Section in its options (the first control of the window always starts a new section, even if that word isn't specified in its options). For example: MyGui := Gui() MyGui.Add ("Edit", "w600") ; Add a fairly wide edit control at the top of the window. MyGui.Add("Text", "section", "First Name:") ; Save this control's position and start a new section. MyGui.Add("Text",, "Last Name:") MyGui.Add("Edit", "ys") ; Start a new column within this section. MyGui.Add("Edit") MyGui.Show xs and ys may optionally be followed by a plus/minus sign and a number. Also, it is possible to specify both the word Section and xs/ys in a control's options; this uses the previous section for itself but establishes a new section for subsequent controls. Omitting either X, Y or both is useful to make a GUI layout automatically adjust to any future changes you might make to the size of controls or font. By contrast, specifying an absolute position for every control might require you to manually shift the position of all controls that lie beneath and/or to the right of a control that is being enlarged or reduced. If both X and Y are omitted, the control will be positioned beneath the previous control using a standard padding distance (the current margin). Consecutive Text or Link controls are given additional vertical padding, so that they typically align better in cases where a column of Edit, DDL or similar-sized controls are later added to their right. To use only the standard vertical margin, specify Y+M or any value for X. If only one component is omitted, its default value depends on which option was used to specify the other component: Specified XDefault for Y xn or xmBeneath all previous controls (maximum Y extent plus margin). xsBeneath all previous controls since the most recent use of the Section option. x+n or xp+nonzeroSame as the previous control's top edge (yp). xp or xp+0Below the previous control (bottom edge plus margin). Specified YDefault for X yn or ymTo the right of all previous controls (maximum X extent plus margin). ysTo the right of all previous controls since the most recent use of the Section option. y+n or yp+nonzeroSame as the previous control's left edge (xp). yp or yp+0To the right of the previous control (right edge plus margin). Storing and Responding to User Input V: Sets the control's Name. Specify the name immediately after the letter V, which is not included in the name. For example, specifying vMyEdit would name the control "MyEdit". Events: Event handlers (such as a function which is called automatically when the user clicks or changes a control) cannot be set within the control's Options. Instead, OnEvent can be used to register a callback function or method for each event of interest. Controls: Common Styles and Other Options Note: In the absence of a preceding sign, a plus sign is assumed; for example, Wrap is the same as +Wrap. By contrast, -Wrap would remove the word-wrapping property. AltSubmit: Uses alternate submit method. For DropDownList, ComboBox, ListBox and Tab, this causes Gui.Submit to store the position of the selected item rather than its text. If no item is selected, a ComboBox will still store the text of its edit field. C: Color of text (has no effect on buttons and status bars). Specify the letter C followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: cRed, cFF2211, c0xFF2211, cDefault. Disabled: Makes an input-capable control appear in a disabled state, which prevents the user from focusing or modifying its contents. Use GuiCtrl.Enabled to enable it later. Note: To make an Edit control read-only, specify the string ReadOnly instead. Also, the word Disabled may optionally be followed immediately by a 0 or 1 to indicate the starting state (0 for enabled and 1 for disabled). In other words, Disabled and "Disabled" VarContainingOne are the same. Hidden: The control is initially invisible. Use GuiCtrl.Visible to show it later. The word Hidden may optionally be followed immediately by a 0 or 1 to indicate the starting state (0 for visible and 1 for hidden). In other words, Hidden and "Hidden" VarContainingOne are the same. Left: Left-justifies the control's text within its available width. This option affects the following controls: Text, Edit, Button, Checkbox, Radio, UpDown, Slider, Tab, Tab2, GroupBox, DateTime. Right: Right-justifies the control's text within its available width. For checkboxes and radio buttons, this also puts the box itself on the right side of the control rather than the left. This option affects the following controls: Text, Edit, Button, Checkbox, Radio, UpDown, Slider, Tab, Tab2, GroupBox, DateTime, Link. Center: Centers the control's text within its available width. This option affects the following controls: Text, Edit, Button, Checkbox, Radio, Slider, GroupBox. Section: Starts a new section and saves this control's position for later use with the xs and ys positioning options described above. Tabstop: Use -Tabstop (i.e. minus Tabstop) to have an input-capable control skipped over when the user presses Tab to navigate. Wrap: Enables word-wrapping of the control's contents within its available width. Since nearly all control types start off with word-wrapping enabled, use -Wrap to disable word-wrapping. VScroll: Provides a vertical scroll bar if appropriate for this type of control. HScroll: Provides a horizontal scroll bar if appropriate for this type of control. The rest of this paragraph applies to

ListBox only. The horizontal scrolling width defaults to 3 times the width of the ListBox. To specify a different scrolling width, include a number immediately after the word HScroll. For example, HScroll500 would allow 500 pixels of scrolling inside the ListBox. However, if the specified scrolling width is smaller than the width of the ListBox, no scroll bar will be shown (though the mere presence of HScroll makes it possible for the horizontal scroll bar to be added later via MyScrollBar.Opt("+HScroll500"), which is otherwise impossible). Controls: Uncommon Styles and Options BackgroundTrans: Uses a transparent background, which allows any control that lies behind a Text, Picture, or GroupBox control to show through. For example, a transparent Text control displayed on top of a Picture control would make the text appear to be part of the picture. Use GuiCtrl.Opt("+Background") to remove this option later. See Picture control's AltSubmit section for more information about transparent images. Known limitation: BackgroundTrans might not work properly for controls inside a Tab control that contains a ListView. If a control type does not support this option, an error is thrown. BackgroundColor: Changes the background color of the control. Replace Color with a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: BackgroundSilver, BackgroundFFDD99. If this option is not present, a Text, Picture, GroupBox, CheckBox, Radio, Slider, Tab or Link control initially defaults to the background color set by Gui.BackColor (or if none or other control type, the system's default background color). Specifying BackgroundDefault or -Background applies the system's default background color. For example, a control can be restored to the default color via LV.Opt("+BackgroundDefault"). Using +Background without specifying a color reverts -Background. If a control type does not support this option, an error is thrown. Border: Provides a thin-line border around the control. Most controls do not need this because they already have a type-specific border. When adding a border to an existing control, it might be necessary to increase the control's width and height by 1 pixel. Redraw: When used with Opt, this option enables or disables redraw (visual updates) for a control by sending it a WM_SETREDRAW message. See Redraw for more details. Theme: This option can be used to override the window's current theme setting for the newly created control. It has no effect when used on an existing control; however, this may change in a future version. See GUI's +/-Theme option for details. (Unnamed Style): Specify a plus or minus sign followed immediately by a decimal or hexadecimal style number. If the sign is omitted, a plus sign is assumed. (Unnamed ExStyle): Specify a plus or minus sign followed immediately by the letter E and a decimal or hexadecimal extended style number. If the sign is omitted, a plus sign is assumed. For example, E0x200 would add the WS_EX_CLIENTEDGE style, which provides a border with a sunken edge that might be appropriate for pictures and other controls. For other extended styles not documented here (since they are rarely used), see Extended Window Styles | Microsoft Docs for a complete list. TextDepending on the specified control type, a string, number or an array. Destroy Removes the window and all its controls, freeing the corresponding memory and system resources. MyGui.Destroy() If MyGui.Destroy() is not used, the window is automatically destroyed when the Gui object is deleted (see General Remarks for details). All GUI windows are automatically destroyed when the script exits. Flash Blinks the window's button in the taskbar. MyGui.Flash(Blink) Blink Type: Boolean If this parameter is omitted or 1 (true), the window's button in the taskbar will blink. This is done by inverting the color of the window's title bar and/or taskbar button (if it has one). Specify 0 (false) to restore the original colors of the title bar and taskbar button (but the actual behavior might vary depending on OS version). In the below example, the window will blink three times because each pair of flashes inverts then restores its appearance: Loop 6 { MyGui.Flash Sleep 500 ; It's quite sensitive to this value; altering it may change the behavior in unexpected ways. } GetClientPos Retrieves the position and size of the window's client area. MyGui.GetClientPos (&X, &Y, &Width, &Height) Each parameter should be a reference to the variable in which to store the respective coordinate. The coordinates are the upper left corner of the window's client area, which is the area not including title bar, menu bar, and borders. X and Y are in screen coordinates. Width is the horizontal distance between the left and right side of the client area, and height the vertical distance between the top and bottom side (in pixels). Unlike WinGetClientPos, this method applies DPI scaling to Width and Height (unless the -DPIScale option was used). GetPos Retrieves the position and size of the window. MyGui.GetPos(&X, &Y, &Width, &Height) Each parameter should be a reference to the variable in which to store the respective coordinate. The coordinates are the upper left corner of the window. X and Y are in screen coordinates. Width is the horizontal distance between the left and right side of the window, and height the vertical distance between the top and bottom side (in pixels). Unlike WinGetPos, this method applies DPI scaling to Width and Height (unless the -DPIScale option was used). Hide Hides the window. MyGui.Hide() Maximize Unhides the window (if necessary) and maximizes it. MyGui.Maximize() Minimize Unhides the window (if necessary) and minimizes it. MyGui.Minimize() Move Moves and/or resizes the GUI window. MyGui.Move(X, Y, Width, Height) X, Y Type: Integer The new position, in screen coordinates. Width, Height Type: Integer The new size. Unlike WinMove, this method applies DPI scaling to Width and Height (unless the -DPIScale option was used). Examples: MyGui.Move(10, 20, 200, 100) MyGui.Move(VarX+10, VarY+5, VarW*2, VarH*1.5) ; Expand the left and right side by 10 pixels. MyGui.GetPos(&x,, &w) MyGui.Move(x-10,, w+20) OnEvent Registers a function or method to be called when the given event is raised. MyGui.OnEvent (EventName, Callback , AddRemove) See OnEvent for details. Opt Sets one or more options for the GUI window. MyGui.Opt(Options) Options Type: String For performance reasons, it is better to set all options in a single line, and to do so before creating the window (that is, before any use of other methods such as Gui.Add). The effect of this parameter is cumulative; that is, it alters only those settings that are explicitly specified, leaving all the others unchanged. Specify a plus sign to add the option and a minus sign to remove it. For example: MyGui.Opt("+Resize -MaximizeBox"). AlwaysOnTop: Makes the window stay on top of all other windows, which is the same effect as WinSetAlwaysOnTop. Border: Provides a thin-line border around the window. This is not common. Caption (present by default): Provides a title bar and a thick window border/edge. When removing the caption from a window that will use WinSetTransColor, remove it only after setting the TransColor. Disabled: Disables the window, which prevents the user from interacting with its controls. This is often used on a window that owns other windows (see Owner). DPIScale: Use MyGui.Opt("-DPIScale") to disable DPI scaling, which is enabled by default. If DPI scaling is enabled, coordinates and sizes passed to or retrieved from the Gui and GuiControl methods/properties are automatically scaled based on screen DPI. For example, with a DPI of 144 (150%), MyGui.Show("w100") would make the Gui 150 (100 * 1.5) pixels wide, and resizing the window to 200 pixels wide via the mouse or WinMove would cause MyGui.GetClientPos(„&W) to set W to 133 (200 // 1.5). A_ScreenDPI contains the system's current DPI. DPI scaling only applies to the Gui and GuiControl methods/properties, so coordinates coming directly from other sources such as ControlGetPos or WinGetPos will not work. There are a number of ways to deal with this: Avoid using hard-coded coordinates wherever possible. For example, use the xp, xs, xm and x+m options for positioning controls and specify height in rows of text instead of pixels. Enable (MyGui.Opt("+DPIScale")) and disable (MyGui.Opt("-DPIScale")) scaling on the fly, as needed. Changing the setting does not affect positions or sizes which have already been set. Manually scale the coordinates. For example, x*(A_ScreenDPI/96) converts x from logical/GUI coordinates to physical/non-GUI coordinates. LastFound: Sets the window to be the last found window (though this is unnecessary in a GUI thread because it is done automatically), which allows functions such as WinGetStyle and WinSetTransparent to operate on it even if it is hidden (that is, DetectHiddenWindows is not necessary). This is especially useful for changing the properties of the window before showing it. For example: MyGui.Opt("+LastFound") WinSetTransColor (CustomColor " 150", MyGui) MyGui.Show() MaximizeBox: Enables the maximize button in the title bar. This is also included as part of Resize below. MinimizeBox (present by default): Enables the minimize button in the title bar. MinSize and MaxSize: Determines the minimum and/or maximum size of the window, such as when the user drags its edges to resize it. Specify the word MinSize and/or MaxSize with no suffix to use the window's current size as the limit (if the window has no current size, it will use the size from the first use of Gui.Show). Alternatively, append the width, followed by an X, followed by the height; for example: MyGui.Opt("+Resize +MinSize640x480"). The dimensions are in pixels, and they specify the size of the window's client area (which excludes borders, title bar, and menu bar). Specify each number as decimal, not hexadecimal. Either the width or the height may be omitted to leave it unchanged (e.g. +MinSize640x or +MinSizex480). Furthermore, Min/MaxSize can be specified more than once to use the window's current size for one dimension and an explicit size for the other. For example, +MinSize +MinSize640x would use the window's current size for the height and 640 for the width. If MinSize and MaxSize are never used, the operating system's defaults are used (similarly, MyGui.Opt("-MinSize -MaxSize") can be used to return to the defaults). Note: the window must have +Resize to allow resizing by the user. OwnDialogs: MyGui.Opt("+OwnDialogs") should be specified in each thread (such as a event handling function of a Button control) for which subsequently displayed MsgBox, InputBox, FileSelect, and DirSelect dialogs should be owned by the window. Such dialogs are modal, meaning that the user cannot interact with the GUI window until dismissing the dialog. By contrast, ToolTip do not become modal even though they become owned; they will merely stay always on top of their owner. In either case, any owned dialog or window is automatically destroyed when its GUI window is destroyed. There is typically no need to turn this setting back off because it does not affect other threads. However, if a thread needs to display both owned and unowned dialogs, it may turn off this setting via MyGui.Opt("-OwnDialogs"). Owner: Use +Owner to make the window owned by another. An owned window has no taskbar button by default, and when visible it is always on top of its owner. It is also automatically destroyed when its owner is destroyed. +Owner can be used before or after the owned window is created. There are two ways to use +Owner, as shown below: MyGui.Opt("+Owner" OtherGui.Hwnd) ; Make the GUI owned by OtherGui. MyGui.Opt ("+Owner") ; Make the GUI owned by the script's main window to prevent display of a taskbar button. +Owner can be immediately followed by the HWND of any top-level window. To prevent the user from interacting with the owner while one of its owned window is visible, disable the owner via MyGui.Opt("+Disabled"). Later (when the time comes to cancel or destroy the owned window), re-enable the owner via MyGui.Opt("-Disabled"). Do this prior to cancel/destroy so that the owner will be reactivated automatically. Parent: Use +Parent immediately followed by the HWND of any window or control to use it as the parent of this window. To convert the GUI back into a top-level window, use -Parent. This option works even after the window is created. Known limitation: Running with UI access prevents the +Parent option from working on an existing window if the new parent is always-on-top and the child window is not. Resize: Makes the window resizable and enables its maximize button in the title bar. To avoid enabling the maximize button, specify +Resize -MaximizeBox. SysMenu (present by default): Specify -SysMenu (minus SysMenu) to omit the system menu and icon in the window's upper left corner. This will also omit the minimize, maximize, and close buttons in the title bar. Theme: By specifying -Theme, all subsequently created controls in the window will have the Classic Theme appearance. To later create additional controls that obey the current theme, turn it back on via +Theme. Note: This option has no effect if the Classic Theme is in effect. Finally, this setting

may be changed for an individual control by specifying +Theme or -Theme in its options when it is created. ToolWindow: Provides a narrower title bar but the window will have no taskbar button. This always hides the maximize and minimize buttons, regardless of whether the WS_MAXIMIZEBOX and WS_MINIMIZEBOX styles are present. (Unnamed Style): Specify a plus or minus sign followed immediately by a decimal or hexadecimal style number. (Unnamed ExStyle): Specify a plus or minus sign followed immediately by the letter E and a decimal or hexadecimal extended style number. For example, +E0x40000 would add the WS_EX_APPWINDOW style, which provides a taskbar button for a window that would otherwise lack one. For other extended styles not documented here (since they are rarely used), see Extended Window Styles | Microsoft Docs for a complete list. Restore Unhides the window (if necessary) and restores it, if it was minimized or maximized beforehand. MyGui.Restore() SetFont Sets the font typeface, size, style, and/or color for controls added to the window from this point onward. Note: Omit both parameters to restore the font to the system's default GUI typeface, size, and color. Otherwise, any font attributes which are not specified will be copied from the previous font. MyGui.SetFont(Options, FontName) Options Type: String Zero or more options. Each option is either a single letter immediately followed by a value, or a single word. To specify more than one option, include a space between each. For example: cBlue s12 bold. The following words are supported: bold, italic, strike, underline, and norm. Norm returns the font to normal weight/boldness and turns off italic, strike, and underline (but it retains the existing color and size). It is possible to use norm to turn off all attributes and then selectively turn on others. For example, specifying norm italic would set the font to normal then to italic. C: Color name (see color chart) or RGB value -- or specify the word Default to return to the system's default color (black on most systems). Example values: cRed, cFFFFAA, cDefault. Note: Buttons and status bars do not obey custom colors. Also, an individual control can be created with a font color other than the current one by including the C option. For example: MyGui.Add("Text", "cRed", "My Text"). S: Size (in points). For example: s12 (specify decimal, not hexadecimal) W: Weight (boldness), which is a number between 1 and 1000 (400 is normal and 700 is bold). For example: w600 (specify decimal, not hexadecimal) Q: Text rendering quality. For example: q3. Q should be followed by a number from the following table: Number Windows Constant Description 0 DEFAULT_QUALITY Appearance of the font does not matter. 1 DRAFT_QUALITY Appearance of the font is less important than when the PROOF_QUALITY value is used. 2 PROOF_QUALITY Character quality of the font is more important than exact matching of the logical-font attributes. 3 NONANTIALIASED_QUALITY Font is never antialiased, that is, font smoothing is not done. 4 ANTIALIASED_QUALITY Font is antialiased, or smoothed, if the font supports it and the size of the font is not too small or too large. 5 CLEARTYPE_QUALITY If set, text is rendered (when possible) using ClearType antialiasing method. For more details of what these values mean, see Microsoft Docs: CreateFont. Since the highest quality setting is usually the default, this feature is more typically used to disable anti-aliasing in specific cases where doing so makes the text clearer. FontName Type: String FontName may be the name of any font, such as one from the font table. If FontName is omitted or does not exist on the system, the previous font's typeface will be used (or if none, the system's default GUI typeface). This behavior is useful to make a GUI window have a similar font on multiple systems, even if some of those systems lack the preferred font. For example, by using the following methods in order, Verdana will be given preference over Arial, which in turn is given preference over MS sans serif: MyGui.SetFont(, "MS sans serif") MyGui.SetFont(, "Arial") MyGui.SetFont(, "Verdana") ; Preferred font. On a related note, the operating system offers standard dialog boxes that prompt the user to pick a font, color, or icon. These dialogs can be displayed via DllCall as demonstrated at GitHub. Show By default, this makes the window visible, unminimizes it (if necessary) and activates it. MyGui.Show(Options) Options Type: String Omit the X, Y, W, and H options below to have the window retain its previous size and position. If there is no previous position, the window will be auto-centered in one or both dimensions if the X and/or Y options mentioned below are absent. If there is no previous size, the window will be auto-sized according to the size and positions of the controls it contains. Zero or more of the following strings may be present in Options (specify each number as decimal, not hexadecimal): Wn: Specify for n the width (in pixels) of the window's client area (the client area excludes the window's borders, title bar, and menu bar). Hn: Specify for n the height of the window's client area, in pixels. Xn: Specify for n the window's X-position on the screen, in pixels. Position 0 is the leftmost column of pixels visible on the screen. Yn: Specify for n the window's Y-position on the screen, in pixels. Position 0 is the topmost row of pixels visible on the screen. Center: Centers the window horizontally and vertically on the screen. xCenter: Centers the window horizontally on the screen. For example: MyGui.Show("xCenter y0"). yCenter: Centers the window vertically on the screen. AutoSize: Resizes the window to accommodate only its currently visible controls. This is useful to resize the window after new controls are added, or existing controls are resized, hidden, or unhidden. For example: MyGui.Show("AutoSize Center"). One of the following may also be present: Minimize: Minimizes the window and activates the one beneath it. Maximize: Maximizes and activates the window. Restore: Unminimizes or unmaximizes the window, if necessary. The window is also shown and activated, if necessary. NoActivate: Unminimizes or unmaximizes the window, if necessary. The window is also shown without activating it. NA: Shows the window without activating it. If the window is minimized, it will stay that way but will probably rise higher in the z-order (which is the order seen in the alt-tab selector). If the window was previously hidden, this will probably cause it to appear on top of the active window even though the active window is not deactivated. Hide: Hides the window and activates the one beneath it. This is identical in function to Gui.Hide except that it allows a hidden window to be moved or resized without showing it. For example: MyGui.Show("Hide x55 y66 w300 h200"). Submit Collects the values from named controls and composes them into an Object. Optionally hides the window. NamedCtrlContents := MyGui.Submit(Hide) Hide Type: Boolean If omitted or 1 (true), the window will be hidden. If 0 (false), the window will not be hidden. The returned object contains one own property per named control, like NamedCtrlContents.%GuiCtrl.Name% := GuiCtrl.Value, with the exceptions noted below. Only input-capable controls which support GuiCtrl.Value and have been given a name are included. Use NamedCtrlContents.NameOfControl to retrieve an individual value or OwnProps to enumerate them all. For DropDownList, ComboBox, ListBox and Tab, the text of the selected item/tab is stored instead of its position number if the control lacks the AltSubmit option, or if the ComboBox's text does not match a list item. Otherwise, Value (the item's position number) is stored. If only one Radio button in a radio group has a name, Submit stores the number of the currently selected button instead of the control's Value. 1 is the first radio button (according to original creation order), 2 is the second, and so on. If there is no button selected, 0 is stored. Excluded because they are not input-capable: Text, Pic, GroupBox, Button, Progress, Link, StatusBar. Also excluded: ListView, TreeView, ActiveX, Custom. __Enum Enumerates the GUI's controls. For Ctrl in MyGui For Hwnd, Ctrl in MyGui Returns a new enumerator. This method is typically not called directly. Instead, the Gui object is passed directly to a for-loop, which calls __Enum once and then calls the enumerator once for each iteration of the loop. Each call to the enumerator returns the next control. The for-loop's variables correspond to the enumerator's parameters, which are: Hwnd Type: Integer The control's HWND. This is present only in the two-parameter mode. Ctrl Type: GuiControl The control's object. For example: For Hwnd, GuiCtrlObj in MyGui MsgBox "Control #" A_Index " is " GuiCtrlObj.ClassNN __New Constructs a new Gui instance. MyGui.__New(Options, Title, EventObj) A Gui subclass may override __New and call super.__New(Options, Title, this) to handle its own events. In such cases, events for the main window (such as Close) do not pass an explicit Gui parameter, as this already contains a reference to the Gui. An exception is thrown if the window has already been constructed or destroyed. Properties BackColor Retrieves or sets the background color of the window. RetrievedColor := MyGui.BackColor MyGui.BackColor := NewColor RetrievedColor is a 6-digit RGB value of the current color previously set by this property, or an empty string if the default color is being used. NewColor is one of the 16 primary HTML color names, a hexadecimal RGB color value (the 0x prefix is optional), a pure numeric RGB color value, or the word Default (or an empty string) for its default color. Example values: "Silver", "FFFFAA", 0xFFFFAA, "Default", "". By default, the window's background color is the system's color for the face of buttons. The color of the menu bar and its submenus can be changed as in this example: MyMenuBar.SetColor "White". To make the background transparent, use WinSetTransColor. However, if you do this without first having assigned a custom window via Gui.BackColor, buttons will also become transparent. To prevent this, first assign a custom color and then make that color transparent. For example: MyGui.BackColor := "EEAA99" WinSetTransColor("EEAA99", MyGui) To additionally remove the border and title bar from a window with a transparent background, use the following: MyGui.Opt("-Caption") To illustrate the above, there is an example of an on-screen display (OSD) near the bottom of this page. FocusedCtrl Retrieves the GuiControl object of the GUI's focused control. GuiCtrlObj := MyGui.FocusedCtrl Note: To be effective, the window generally must not be minimized or hidden. Hwnd Retrieves the window handle (HWND) of the GUI window. CurrentHwnd := MyGui.Hwnd A GUI's HWND is often used with PostMessage, SendMessage, and DllCall. It can also be used directly in a WinTitle parameter. MarginX Retrieves or sets the size of horizontal margins between sides and subsequently created controls. RetrievedValue := MyGui.MarginX MyGui.MarginX := NewValue RetrievedValue is the number of pixels of the current horizontal margin. NewValue is the number of pixels of space to leave at the left and right side of the window when auto-positioning any control that lacks an explicit X coordinate. Also, the margin is used to determine the horizontal distance that separates auto-positioned controls from each other. Finally, the margin is taken into account by the first use of Gui.Show to calculate the window's size (when no explicit size is specified). By default, this margin is proportional to the size of the currently selected font (1.25 times font-height for left & right). MarginY Retrieves or sets the size of vertical margins between sides and subsequently created controls. RetrievedValue := MyGui.MarginY MyGui.MarginY := NewValue RetrievedValue is the number of pixels of the current vertical margin. NewValue is the number of pixels of space to leave at the top and bottom side of the window when auto-positioning any control that lacks an explicit Y coordinate. Also, the margin is used to determine the vertical distance that separates auto-positioned controls from each other. Finally, the margin is taken into account by the first use of Gui.Show to calculate the window's size (when no explicit size is specified). By default, this margin is proportional to the size of the currently selected font (0.75 times font-height for top & bottom). MenuBar Retrieves or sets the window's menu bar. MyGui.MenuBar := Bar Bar := MyGui.MenuBar Bar is a MenuBar object created by MenuBar(). For example: FileMenu := Menu() FileMenu.Add "&Open`tCtrl+O", (*) => FileSelect() ; See remarks below about Ctrl+O. FileMenu.Add "E&xit", (*) => ExitApp() HelpMenu := Menu() HelpMenu.Add "&About", (*) => MsgBox("Not implemented") Menus := MenuBar() Menus.Add "&File", FileMenu ; Attach the two submenus that were created above. Menus.Add "&Help", HelpMenu MyGui := Gui() MyGui.MenuBar := Menus MyGui.Show "w300 h200" In the first line above, notice that &Open is followed by Ctrl+O (with a tab character in between). This indicates a keyboard shortcut that the user may press instead of selecting the menu item. If the shortcut uses only the standard modifier key names Ctrl, Alt and Shift, it is automatically registered as a keyboard

accelerator for the GUI. Single-character accelerators with no modifiers are case-sensitive and can be triggered by unusual means such as IME or Alt+NNNN. If a particular key combination does not work automatically, the use of a context-sensitive hotkey may be required. However, such hotkeys typically cannot be triggered by Send and are more likely to interfere with other scripts than a standard keyboard accelerator. To remove a window's current menu bar, use MyGui.MenuBar := "" (that is, assign an empty string). Name Retrieves or sets a custom name for the GUI window. RetrievedName := MyGui.Name MyGui.Name := NewName Title Retrieves or sets the GUI's title. RetrievedTitle := MyGui.Title MyGui.Title := NewTitle __Item Retrieves the GuiControl object associated with the specified name, text, ClassNN or HWND. GuiCtrlObj := Gui[Name] Keyboard Navigation A GUI window may be navigated via Tab, which moves keyboard focus to the next input-capable control (controls from which the Tabstop style has been removed are skipped). The order of navigation is determined by the order in which the controls were originally added. When the window is shown for the first time, the first input-capable control that has the Tabstop style (which most control types have by default) will have keyboard focus, unless that control is a Button and there is a Default button, in which case the latter is focused instead. Certain controls may contain an ampersand (&) to create a keyboard shortcut, which might be displayed in the control's text as an underlined character (depending on system settings). A user activates the shortcut by holding down Alt then typing the corresponding character. For buttons, checkboxes, and radio buttons, pressing the shortcut is the same as clicking the control. For GroupBoxes and Text controls, pressing the shortcut causes keyboard focus to jump to the first input-capable tabstop control that was created after it. However, if more than one control has the same shortcut key, pressing the shortcut will alternate keyboard focus among all controls with the same shortcut. To display a literal ampersand inside the control types mentioned above, specify two consecutive ampersands as in this example: MyGui.Add("Button",, "Save && Exit"). Window Appearance For its icon, a GUI window uses the tray icon that was in effect at the time the window was created. Thus, to have a different icon, change the tray icon before creating the window. For example: TraySetIcon("MyIcon.ico"). It is also possible to have a different large icon for a window than its small icon (the large icon is displayed in the alt-tab task switcher). This can be done via LoadPicture and SendMessage; for example: iconsize := 32 ; Ideal size for alt-tab varies between systems and OS versions. hIcon := LoadPicture("My Icon.ico", "Icon1 w" iconsize " h" iconsize, &imgtype) MyGui := Gui() SendMessage(0x0080, 1, hIcon, MyGui) ; 0x0080 is WM_SETICON; and 1 means ICON_BIG (vs. 0 for ICON_SMALL). MyGui.Show() Due to OS limitations, Checkboxes, Radio buttons, and GroupBoxes for which a non-default text color was specified will take on the Classic Theme appearance. Related topic: window's margin. General Remarks Use the GuiControl object to operate upon individual controls in a GUI window. Each GUI window may have up to 11,000 controls. However, use caution when creating more than 5000 controls because system instability may occur for certain control types. The GUI window is automatically destroyed when the Gui object is deleted, which occurs when its reference count reaches zero. However, this does not typically occur while the window is visible, as Show automatically increments the reference count. While the window is visible, the user can interact with it and raise events which are handled by the script. When the user closes the window or it is hidden by Hide, Show or Submit, this extra reference is released. To keep a GUI window "alive" without calling Show or retaining a reference to its Gui object, the script can increment the object's reference count with ObjAddRef (in which case ObjRelease must be called when the window is no longer needed). For example, this might be done when using a hidden GUI window to receive messages, or if the window is shown by "external" means such as WinShow (by this script or any other). If the script is not persistent for any other reason, it will exit after the last visible GUI is closed; either when the last thread completes or immediately if no threads are running. Related GuiControl object, GuiFromHwnd, GuiCtrlFromHwnd, Control Types, ListView, TreeView, Menu object, Control functions, MsgBox, FileSelect, DirSelect Examples Creates a popup window. MyGui := Gui(, "Title of Window") MyGui.Opt("+AlwaysOnTop +Disabled -SysMenu +Owner") ; +Owner avoids a taskbar button. MyGui.Add("Text",, "Some text to display.") MyGui.Show("NoActivate") ; NoActivate avoids deactivating the currently active window. Creates a simple input-box that asks for the first and last name. MyGui := Gui(, "Simple Input Example") MyGui.Add("Text",, "First name:") MyGui.Add("Text",, "Last name:") MyGui.Add("Edit", "vFirstName ym") ; The ym option starts a new column of controls. MyGui.Add("Edit", "vLastName") MyGui.Add("Button", "default", "OK").OnEvent("Click", ProcessUserInput) MyGui.OnEvent("Close", ProcessUserInput) MyGui.Show() ProcessUserInput(*) { Saved := MyGui.Submit() ; Save the contents of named controls into an object. MsgBox("You entered '" Saved.FirstName " " Saved.LastName "'.") } Creates a tab control with multiple tabs, each containing different controls to interact with. MyGui := Gui() Tab := MyGui.Add("Tab3",, ["First Tab","Second Tab","Third Tab"]) MyGui.Add("Checkbox", "vMyCheckbox", "Sample checkbox") Tab.UseTab(2) MyGui.Add("Radio", "vMyRadio", "Sample radio1") MyGui.Add("Radio",, "Sample radio2") Tab.UseTab(3) MyGui.Add("Edit", "vMyEdit r5") ; r5 means 5 rows tall. Tab.UseTab() ; i.e. subsequently-added controls will not belong to the tab control. Btn := MyGui.Add("Button", "default xm", "OK") ; xm puts it at the bottom left corner. Btn.OnEvent("Click", ProcessUserInput) MyGui.OnEvent("Close", ProcessUserInput) MyGui.OnEvent("Escape", ProcessUserInput) MyGui.Show() ProcessUserInput(*) { Saved := MyGui.Submit() ; Save the contents of named controls into an object. MsgBox("You entered:`n" Saved.MyCheckbox "`n" Saved.MyRadio "`n" Saved.MyEdit) } Creates a ListBox control containing files in a directory. MyGui := Gui() MyGui.Add("Text",, "Pick a file to launch from the list below.") LB := MyGui.Add("ListBox", "w640 r10") LB.OnEvent("DoubleClick", LaunchFile) Loop Files, "C:*.*" ; Change this folder and wildcard pattern to suit your preferences. LB.Add([A_LoopFilePath]) MyGui.Add("Button", "Default", "OK").OnEvent("Click", LaunchFile) MyGui.Show() LaunchFile(*) { if MsgBox("Would you like to launch the file or document below?`n`n" LB.Text,, 4) = "No" return ; Otherwise, try to launch it: try Run(LB.Text) if A_LastError MsgBox("Could not launch the specified file. Perhaps it is not associated with anything.") } Displays a context-sensitive help (via ToolTip) whenever the user moves the mouse over a particular control. MyGui := Gui() MyEdit := MyGui.Add("Edit") ; Store the tooltip text in a custom property: MyEdit.ToolTip := "This is a tooltip for the control whose name is MyEdit." MyDDL := MyGui.Add("DropDownList",, ["Red","Green","Blue"]) MyDDL.ToolTip := "Choose a color from the drop-down list." MyGui.Add("Checkbox",, "This control has no tooltip.") MyGui.Show() OnMessage(0x0200, On_WM_MOUSEMOVE) On_WM_MOUSEMOVE(wParam, lParam, msg, Hwnd) { static PrevHwnd := 0 if (Hwnd != PrevHwnd) { Text := "", ToolTip() ; Turn off any previous tooltip. CurrControl := GuiCtrlFromHwnd(Hwnd) if CurrControl { if !CurrControl.HasProp("ToolTip") return ; No tooltip for this control. Text := CurrControl.ToolTip SetTimer () => ToolTip(Text), -1000 SetTimer () => ToolTip(), -4000 ; Remove the tooltip. } PrevHwnd := Hwnd } } Creates an On-screen display (OSD) via transparent window. MyGui := Gui() MyGui.Opt("+AlwaysOnTop -Caption +ToolWindow") ; +ToolWindow avoids a taskbar button and an alt-tab menu item. MyGui.BackColor := "EEAA99" ; Can be any RGB color (it will be made transparent below). MyGui.SetFont("s32") ; Set a large font size (32-point). CoordText := MyGui.Add("Text", "cLime", "XXXXX YYYYY") ; XX & YY serve to auto-size the window. ; Make all pixels of this color transparent and make the text itself translucent (150): WinSetTransColor(MyGui.BackColor " 150", MyGui) SetTimer(UpdateOSD, 200) UpdateOSD() ; Make the first update immediate rather than waiting for the timer. MyGui.Show("x0 y400 NoActivate") ; NoActivate avoids deactivating the currently active window. UpdateOSD(*) { MouseGetPos &MouseX, &MouseY CoordText.Value := "X" MouseX ", Y" MouseY } Creates a moving progress bar overlayed on a background image. MyGui := Gui() MyGui.BackColor := "White" MyGui.Add("Picture", "x0 y0 h350 w450", A_WinDir "\Web\Wallpaper\Windows\img0.jpg") MyBtn := MyGui.Add("Button", "Default xp+20 yp+250", "Start the Bar Moving") MyBtn.OnEvent("Click", MoveBar) MyProgress := MyGui.Add("Progress", "w416") MyText := MyGui.Add("Text", "wp") ; wp means "use width of previous". MyGui.Show() MoveBar(*) { Loop Files, A_WinDir "*.*" { if (A_Index > 100) break MyProgress.Value := A_Index MyText.Value := A_LoopFileName Sleep 50 } MyText.Value := "Bar finished." } Creates a simple image viewer. MyGui := Gui("+Resize") MyBtn := MyGui.Add("Button", "default", "&Load New Image") MyBtn.OnEvent("Click", LoadNewImage) MyRadio := MyGui.Add("Radio", "ym+5 x+10 checked", "Load &actual size") MyGui.Add("Radio", "ym+5 x+10", "Load to &fit screen") MyPic := MyGui.Add("Pic", "xm") MyGui.Show() LoadNewImage(*) { Image := FileSelect(,, "Select an image:", "Images (*.gif; *.jpg; *.bmp; *.png; *.tif; *.ico; *.cur; *.ani; *.exe; *.dll)") if Image = "" return if (MyRadio.Value) ; Display image at its actual size. { Width := 0 Height := 0 } else ; Second radio is selected: Resize the image to fit the screen. { Width := A_ScreenWidth - 28 ; Minus 28 to allow room for borders and margins inside. Height := -1 ; "Keep aspect ratio" seems best. } MyPic.Value := Format("*w{1} *h{2} {3}", Width, Height, Image) ; Load the image. MyGui.Title := Image MyGui.Show("xCenter y0 AutoSize") ; Resize the window to match the picture size. } Creates a simple text editor with menu bar. ; Create the MyGui window: MyGui := Gui("+Resize", "Untitled") ; Make the window resizable. ; Create the submenus for the menu bar: FileMenu := Menu() FileMenu.Add("&New", MenuFileNew) FileMenu.Add("&Open", MenuFileOpen) FileMenu.Add("&Save", MenuFileSave) FileMenu.Add("Save &As", MenuFileSaveAs) FileMenu.Add() ; Separator line. FileMenu.Add("E&xit", MenuFileExit) HelpMenu := Menu() HelpMenu.Add("&About", MenuHelpAbout) ; Create the menu bar by attaching the submenus to it: MyMenuBar := MenuBar() MyMenuBar.Add("&File", FileMenu) MyMenuBar.Add("&Help", HelpMenu) ; Attach the menu bar to the window: MyGui.MenuBar := MyMenuBar ; Create the main Edit control: MainEdit := MyGui.Add("Edit", "WantTab W600 R20") ; Apply events: MyGui.OnEvent("DropFiles", Gui_DropFiles) MyGui.OnEvent("Size", Gui_Size) MenuFileNew() ; Apply default settings. MyGui.Show() ; Display the window. MenuFileNew(*) { MainEdit.Value := "" ; Clear the Edit control. FileMenu.Disable("3&") ; Gray out &Save. MyGui.Title := "Untitled" } MenuFileOpen(*) { MyGui.Opt("+OwnDialogs") ; Force the user to dismiss the FileSelect dialog before returning to the main window. SelectedFileName := FileSelect(3,, "Open File", "Text Documents (*.txt)") if SelectedFileName = "" ; No file selected. return global CurrentFileName := readContent(SelectedFileName) } MenuFileSave(*) { saveContent(CurrentFileName) } MenuFileSaveAs(*) { MyGui.Opt("+OwnDialogs") ; Force the user to dismiss the FileSelect dialog before returning to the main window. SelectedFileName := FileSelect("S16",, "Save File", "Text Documents (*.txt)") if SelectedFileName = "" ; No file selected. return global CurrentFileName := saveContent(SelectedFileName) } MenuFileExit(*) ; User chose "Exit" from the File menu. { WinClose() } MenuHelpAbout(*) { About := Gui("+owner" MyGui.Hwnd) ; Make the main window the owner of the "about box". MyGui.Opt("+Disabled") ; Disable main window. About.Add("Text",, "Text for about box.") About.Add("Button", "Default", "OK").OnEvent("Click", About_Close) About.OnEvent("Close", About_Close) About.OnEvent("Escape", About_Close) About.Show() About_Close(*) { MyGui.Opt("-Disabled") ; Re-enable the main window (must be done prior to the next step). About.Destroy() ; Destroy the about box. } } readContent(FileName) { try FileContent := FileRead(FileName) ; Read the file's contents into the variable. catch { MsgBox("Could not open '" FileName "'.") return } MainEdit.Value := FileContent ; Put the text into the control.

FileMenu.Enable("3&") ; Re-enable &Save. MyGui.Title := FileName ; Show file name in title bar. return FileName } saveContent(FileName) { try { if FileExist (FileName) FileDelete(FileName) FileAppend(MainEdit.Value, FileName) ; Save the contents to the file. } catch { MsgBox("The attempt to overwrite '" FileName "' failed.") return } ; Upon success, Show file name in title bar (in case we were called by MenuFileSaveAs): MyGui.Title := FileName return FileName } Gui_DropFiles(thisGui, Ctrl, FileArray, *) ; Support drag & drop. { CurrentFileName := readContent(FileArray[1]) ; Read the first file only (in case there's more than one). } Gui_Size(thisGui, MinMax, Width, Height) { if MinMax = -1 ; The window has been minimized. No action needed. return ; Otherwise, the window has been resized or maximized. Resize the Edit control to match. MainEdit.Move(„ Width-20, Height-20) } GuiControl Object - Methods & Properties | AutoHotkey v2 GuiControl Object class Gui.Control extends Object Provides an interface for modifying GUI controls and retrieving information about them. Gui.Add, Gui.__Item and GuiCtrlFromHwnd return an object of this type. "GuiCtrl" is used below as a placeholder for instances of the Gui.Control class. Gui.Control serves as the base class for all GUI controls, but each type of control has its own class. Some of the following methods are defined by the prototype of the appropriate class, or the Gui.List base class. See Built-in Classes for a full list. In addition to the methods and property inherited from Object, GuiControl objects may have the following predefined methods and properties. Table of Contents Methods: Add: Appends the specified entries at the current list of a ListBox, DropDownList, ComboBox, or Tab control. Choose: Sets the selection in a ListBox, DropDownList, ComboBox, or Tab control to be the specified value. Delete: Deletes the specified entry or all entries of a ListBox, DropDownList, ComboBox, or Tab control. Focus: Sets keyboard focus to the control. GetPos: Retrieves the position and size of the control. Move: Moves and/or resizes the control. OnCommand: Registers a function or method to be called when a control notification is received via the WM_COMMAND message. OnEvent: Registers a function or method to be called when the given event is raised. OnNotify: Registers a function or method to be called when a control notification is received via the WM_NOTIFY message. Opt: Sets various options and styles for the appearance and behavior of the control. Redraw: Redraws the region of the GUI window occupied by the control. SetFont: Sets the font typeface, size, style, and/or color for the control. SetFormat: Sets the display format of a DateTime control. UseTab: Causes subsequently added controls to be belong to the specified tab of a Tab control. Properties: ClassNN: Retrieves the ClassNN of the control. Enabled: Retrieves the current interaction state of the control, or enables or disables (grays out) it. Focused: Retrieves the current focus state of the control. Gui: Retrieves the control's GUI parent. Hwnd: Retrieves the HWND of the control. Name: Retrieves or sets the explicit name of the control. Text: Retrieves or sets the text/caption of the control. Type: Retrieves the type of the control. Value: Retrieves or sets new contents into a value-capable control. Visible: Retrieves the current visibility state of the control, or shows or hides it. General Remarks: Redraw: Performance-related remarks about redraw behaviour of controls. Methods Add Appends list items to a ListBox, DropDownList or ComboBox, or tabs to a Tab control. GuiCtrl.Add(Items) Items Type: Array of Strings An array of strings (e.g. ["Red","Green","Blue"]) to be inserted at the end of the control's list. To replace (overwrite) the list instead, use GuiCtrl.Delete beforehand. Use GuiCtrl.Choose to select an item. Related: ListView.Add, TreeView.Add Choose Sets the selection in a ListBox, DropDownList, ComboBox, or Tab control to be the specified value. GuiCtrl.Choose(Value) Value Type: Integer or String This parameter should be 1 for the first entry, 2 for the second, etc. If Value is a string (even a numeric string), the entry whose leading part matches Value will be selected. The search is not case sensitive. For example, if the control contains the item "UNIX Text", specifying the word unix (lowercase) would be enough to select it. For a multi-select ListBox, all matching items are selected. If Value is zero or empty, the current selection is removed. To select or deselect all items in a multi-select ListBox, follow this example: PostMessage 0x0185, 1, -1, ListBox ; Select all items. 0x0185 is LB_SETSEL. PostMessage 0x0185, 0, -1, ListBox ; Deselect all items. ListBox.Choose(0) ; Deselect all items. Unlike ControlChooseIndex, this method does not raise a Change or DoubleClick event. Delete Deletes the specified entry or all entries of a ListBox, DropDownList, ComboBox, or Tab control. GuiCtrl.Delete(Value) Value Type: Integer This parameter should be 1 for the first entry, 2 for the second, etc. If omitted, all entries are deleted. For tab controls, you should note that a tab's sub-controls stay associated with their original tab number; that is, they are never associated with their tab's actual display-name. For this reason, renaming or removing a tab will not change the tab number to which the sub-controls belong. For example, if there are three tabs ["Red","Green","Blue"] and the second tab is removed by means of MyTab.Delete(2), the sub-controls originally associated with Green will now be associated with Blue. Because of this behavior, only tabs at the end should generally be removed. Tabs that are removed in this way can be added back later, at which time they will reclaim their original set of controls. Related: ListView.Delete, TreeView.Delete Focus Sets keyboard focus to the control. GuiCtrl.Focus() Note: To be effective, the window generally must not be minimized or hidden. GetPos Retrieves the position and size of the control. GuiCtrl.GetPos(X, Y, Width, Height) Each parameter should be a reference to the variable in which to store the respective coordinate. The position is relative to the GUI window's client area, which is the area not including title bar, menu bar, and borders. Unlike ControlGetPos, this method applies DPI scaling to the returned coordinates (unless the -DPIScale option was used). Example: MyEdit.GetPos(&x, &y, &w, &h) MsgBox "The X coordinate is " x ". The Y coordinate is " y ". The width is " w ". The height is " h "." Move Moves and/or resizes the control. GuiCtrl.Move(X, Y, Width, Height) X, Y Type: Integer The new position, relative to the GUI window's client area, which is the area not including title bar, menu bar, and borders. Width, Height Type: Integer The new size. Unlike ControlMove, this method applies DPI scaling to the coordinates (unless the -DPIScale option was used). Examples: MyEdit.Move(10, 20, 200, 100) MyEdit.Move(VarX+10, VarY+5, VarW*2, VarH*1.5) OnCommand Registers a function or method to be called when a control notification is received via the WM_COMMAND message. GuiCtrl.OnCommand(NotifyCode, Callback , AddRemove) See OnCommand for details. OnEvent Registers a function or method to be called when the given event is raised. GuiCtrl.OnEvent(EventName, Callback , AddRemove) See OnEvent for details. OnNotify Registers a function or method to be called when a control notification is received via the WM_NOTIFY message. GuiCtrl.OnNotify(NotifyCode, Callback , AddRemove) See OnNotify for details. Opt Adds or removes various options and styles of the control. GuiCtrl.Opt(Options) Options Type: String Either control-specific or general options and styles. In the following example, the control is disabled and its background is restored to the system default: MyEdit.Opt("+Disabled -Background") In the next example, the OK button is made the new default button: OKButton.Opt("+Default") Although styles and extended styles are also recognized, some of them cannot be applied or removed after a control has been created. Even if a change is successfully applied, the control might choose to ignore it. Redraw Redraws the region of the GUI window occupied by the control. GuiCtrl.Redraw() Although this may cause an unwanted flickering effect when called repeatedly and rapidly, it solves display artifacts for certain control types such as GroupBoxes. SetFont Sets the font typeface, size, style, and/or color for the control. Note: Omit both parameters to set the font to the GUI's current font, as set by Gui.SetFont. Otherwise, any font attributes which are not specified will be copied from the control's previous font. Text color is changed only if specified in Options. GuiCtrl.SetFont(Options, FontName) For details about both parameters, see Gui.SetFont. SetFormat Sets the display format of a DateTime control. GuiCtrl.SetFormat(Format) See SetFormat for details. UseTab Causes subsequently added controls to be belong to the specified tab of a Tab control. GuiCtrl.UseTab(Value, ExactMatch) Value Type: Integer or String This parameter should be 1 for the first tab, 2 for the second, etc. If Value is not an integer, the tab whose leading part matches Value will be used. The search is not case sensitive. For example, if a the control contains the tab "UNIX Text", specifying the word unix (lowercase) would be enough to use it. If Value is zero, a blank string or omitted, subsequently controls are added outside the Tab control. ExactMatch Type: Boolean If this parameter is true, Value has to be an exact match, but not case sensitive. Properties ClassNN Retrieves the ClassNN of the control. ClassNN := GuiCtrl.ClassNN Enabled Retrieves the current interaction state of the control, or enables or disables (grays out) it. RetrievedState := GuiCtrl.Enabled GuiCtrl.Enabled := NewState For Tab controls, this will also enable or disable all of the tab's sub-controls. However, any sub-control explicitly disabled via GuiCtrl.Enabled := false will remember that setting and thus remain disabled even after its Tab control is re-enabled. Focused Retrieves the current focus state of the control. RetrievedState := GuiCtrl.Focused Note: To be effective, the window generally must not be minimized or hidden. Gui Retrieves the Gui object of the control's parent GUI window. GuiObj := GuiCtrl.Gui Hwnd Retrieves the window handle (HWND) of the control. Hwnd := GuiCtrl.Hwnd A control's HWND is often used with PostMessage, SendMessage, and DllCall. Name Retrieves or sets the name of the control. RetrievedName := GuiCtrl.Name GuiCtrl.Name := NewName The control's name can be used with Gui.__Item to retrieve the GuiControl object. For most input-capable controls, the name is also used by Gui.Submit. Text Retrieves or sets the text/caption of the control. RetrievedText := GuiCtrl.Text GuiCtrl.Text := NewText Note: If the control has no visible caption text and no (single) text value, this property corresponds to the control's hidden caption text (like ControlGetText/ControlSetText). Caption/display text: The Text property retrieves or sets the caption/display text of the following control types: Button, Checkbox, Edit, GroupBox, Link, Radio, Text. Since the control will not expand automatically, use GuiCtrl.Move("w300") or similar if the control needs to be widened. DateTime: The Text property retrieves the formatted text displayed by the control. Assigning a formatted date/time string to the control is not supported. To change the date/time being displayed, assign GuiCtrl.Value a date-time stamp in YYYYMMDDHH24MISS format. Edit: As with other controls, the text is retrieved or set as-is; no end-of-line translation is performed. To retrieve or set the text of a multi-line Edit control while also translating between `r`n and `n, use GuiCtrl.Value. StatusBar: The Text property retrieves or sets the text of the first part only (use SB.SetText for greater flexibility). List item text: The Text property retrieves or sets the currently selected item/tab for the following control types: Tab, DropDownList, ComboBox, ListBox. NewText should be the full text (case insensitive) of the item/tab to select. For a ComboBox, if there is no selected item, the text in the control's edit field is retrieved instead. For other controls, RetrievedText is empty/blank. Similarly, if NewText is empty/blank, the current item/tab will be deselected. For a multi-select ListBox, RetrievedText is an array. NewText cannot be an array, but if multiple items match, they are all selected. To select multiple items with different text, call GuiCtrl.Choose repeatedly. To select an item by its position number instead of its text, use GuiCtrl.Value. Type Retrieves the type of the control. RetrievedType := GuiCtrl.Type Depending on the control type, RetrievedType is one of the following strings: Text, Edit, UpDown, Pic, Button, CheckBox, Radio, DDL, ComboBox, ListBox, ListView, TreeView, Link, Hotkey, DateTime, MonthCal, Slider, Progress, GroupBox, Tab, Tab2, Tab3, StatusBar, ActiveX, Custom. Value Retrieves or sets the contents of a control. RetrievedValue := GuiCtrl.Value GuiCtrl.Value := NewValue Note: If the control is not value-capable, RetrievedValue will be blank and assigning NewValue will raise an error. Invalid values throw an exception. Following control types are value-capable: Picture RetrievedValue is the picture's file name as it was originally specified when the Picture control was created. This name does not

change even if a new picture file name is specified. NewValue is the filename (or handle) of the new image to load (see Picture for supported file types). Zero or more of the following options may be specified immediately in front of the filename: *wN (width N), *hN (height N), and *IconN (icon group number N in a DLL or EXE file). In the following example, the default icon from the second icon group is loaded with a width of 100 and an automatic height via "keep aspect ratio": MyPic.Value := "*icon2 *w100 *h-1 C:\My Application.exe". Specify *w0 *h0 to use the image's actual width and height. If *w and *h are omitted, the image will be scaled to fit the current size of the control. When loading from a multi-icon .ICO file, specifying a width and height also determines which icon to load. Note: Use only one space or tab between the final option and the filename itself; any other spaces and tabs are treated as part of the filename. Text RetrievedValue is the text/caption of the Text control. NewValue is the control's new text. Since the control will not expand automatically, use GuiCtrl.Move("w300") if the control needs to be widened. Edit RetrievedValue is the current content of the Edit control. For multi-line controls, any line breaks in the text will be represented as plain linefeeds (`n) rather than the traditional CR+LF (`r`n) used by non-GUI functions such as ControlGetText and ControlSetText. NewValue is the new content. For multi-line controls, Any linefeeds (`n) in NewValue that lack a preceding carriage return (`r) are automatically translated to CR+LF (`r`n) to make them display properly. However, this is usually not a concern because when using Gui.Submit or when retrieving this value this translation will be reversed automatically by replacing CR+LF with LF (`n). To retrieve or set the text without translating `n to or from `r`n, use GuiCtrl.Text. Hotkey RetrievedValue is the modifiers and key name if there is a hotkey in the Hotkey control; otherwise blank. Examples: ^!C, ^Home, +^NumpadHome. NewValue can be a set of modifiers with a key name, or blank to clear the control. Examples: ^!c, ^Numpad1, +Home. The only modifiers supported are ^ (Ctrl), ! (Alt), and + (Shift). See the key list for available key names. Checkbox / Radio RetrievedValue is 1 if the Checkbox or Radio control is checked, 0 if it is unchecked, or -1 if it has a gray checkmark. NewValue can be 0 to uncheck the button, 1 to check it, or -1 to give it a gray checkmark. For Radio buttons, if one is being checked (turned on) and it is a member of a multi-radio group, the other radio buttons in its group will be automatically unchecked. To get or set control's text/caption instead, use GuiCtrl.Text. DateTime / MonthCal RetrievedValue is a date-time stamp in YYYYMMDDHH24MISS format currently selected in the DateTime or MonthCal control. NewValue is a date-time stamp in YYYYMMDDHH24MISS format. Specify A_Now to use the current date and time (today). For DateTime controls, NewValue may be an empty string to cause the control to have no date/time selected (if it was created with that ability). For MonthCal controls, a range may be specified if the control is multi-select. UpDown / Slider / Progress RetrievedValue is the current position of the UpDown, Slider or Progress control. NewValue is the new position of the control, for example MySlider.Value := 10. To adjust the value by a relative amount, use the operators +=, -=, ++ or -- instead of :=. If the new position would be outside the range of the control, the control is generally set to the nearest valid value. Tab / DropDownList / ComboBox / ListBox RetrievedValue is the position of the currently selected item/tab in the Tab, DropDownList, ComboBox or ListBox control. If none is selected, zero is returned. If text is entered into a ComboBox, the first item with matching text is used. If there is no matching item, the result is zero even if there is text in the control. For a multi-select ListBox, the result is an array of numbers (which is empty if no items are selected). NewValue is the position of a single item/tab to select, or zero to clear the current selection (this is valid even for Tab controls). To select multiple items, call GuiCtrl.Choose repeatedly. To get or set the selected item given its text instead of its position, use GuiCtrl.Text. ActiveX RetrievedValue is the ActiveX object of the ActiveX control. For example, if the control was created with the text Shell.Explorer, this is a WebBrowser object. The same wrapper object is returned each time. Visible Retrieves the current visibility state of the control, or shows or hides it. RetrievedState := GuiCtrl.Visible GuiCtrl.Visible := NewState For Tab controls, this will also show or hide all of the tab's sub-controls. If you additionally want to prevent a control's shortcut key (underlined letter) from working, disable the control via GuiCtrl.Enabled := false. General Remarks Redraw When adding a large number of items to a control (such as a ListView, TreeView or ListBox), performance can be improved by preventing the control from being redrawn while the changes are being made. This is done by using GuiCtrl.Opt("-Redraw") before adding the items and GuiCtrl.Opt("+Redraw") afterward. Changes to the control which have not yet become visible prior to disabling redraw will generally not become visible until after redraw is re-enabled. For performance reasons, changes to a control's content generally do not cause the control to be immediately redrawn even if redraw is enabled. Instead, a portion of the control is "invalidated" and is usually repainted after a brief delay, when the program checks its internal message queue. The script can force this to take place immediately by calling Sleep -1. GUI Control Types - Syntax & Usage | AutoHotkey v2 GUI Control Types Table of Contents Text, Edit, UpDown, Picture Button, Checkbox, Radio DropDownList, ComboBox ListBox, ListView, TreeView Link, Hotkey, DateTime MonthCal, Slider, Progress GroupBox, Tab3, StatusBar ActiveX (e.g. Internet Explorer Control) Custom Text Description: A region containing borderless text that the user cannot edit. Often used to label other controls. For example: MyGui.Add("Text",, "Please enter your name:") Or: MyGui.AddText(, "Please enter your name:") Appearance: In this case, the last parameter is the string to display. It may contain linefeeds (`n) to start new lines. In addition, a single long line can be broken up into several shorter ones by means of a continuation section. If a width (W) is specified in Options but no rows (R) or height (H), the text will be word-wrapped as needed, and the control's height will be set automatically. To detect when the user clicks the text, use the Click event. For example: MyGui := Gui() FakeLink := MyGui.Add("Text", "", "Click here to launch Google.") FakeLink.SetFont("underline cBlue") FakeLink.OnEvent ("Click", LaunchGoogle) ; Alternatively, a Link control can be used: MyGui.Add("Link",, 'Click here to launch Google.') MyGui.Show() LaunchGoogle(*) { Run ("www.google.com") } Text controls also support the DoubleClick event. Only Text controls with the SS_NOTIFY (0x100) style send click and double-click notifications, so OnEvent automatically adds this style when a Click or DoubleClick callback is registered. The SS_NOTIFY style causes the OS to automatically copy the control's text to the clipboard when it is double-clicked. An ampersand (&) may be used in the text to underline one of its letters. For example: MyGui.Add ("Text",, "&First Name:") MyGui.Add("Edit") In the example above, the letter F will be underlined, which allows the user to press the shortcut key Alt+F to set keyboard focus to the first input-capable control that was added after the text control. To instead display a literal ampersand, specify two consecutive ampersands (&&). To disable all special treatment of ampersands, include 0x80 in the control's options. See general options for other options like Right, Center, and Hidden. See also: position and sizing of controls. Edit Description: An area where free-form text can be typed by the user. For example: MyGui.Add("Edit", "r9 vMyEdit w135", "Text to appear inside the edit control (omit this parameter to start off empty).") Or: MyGui.AddEdit("r9 vMyEdit w135", "Text to appear inside the edit control (omit this parameter to start off empty).") Appearance: The control will be multi-line if it has more than one row of text. For example, specifying r3 in Options will create a 3-line edit control with the following default properties: a vertical scroll bar, word-wrapping enabled, and Enter captured as part of the input rather than triggering the window's default button. To start a new line in a multi-line edit control, the last parameter (contents) may contain either a solitary linefeed (`n) or a carriage return and linefeed (`r`n). Both methods produce literal `r`n pairs inside the Edit control. However, when the control's content is retrieved via MyGui.Submit or GuiCtrl.Value, each `r`n in the text is always translated to a plain linefeed (`n). To bypass this End-of-Line translation, use GuiCtrl.Text. To write the text to a file, follow this example: FileAppend(MyEdit.Text, "C:\Saved File.txt"). If the control has word-wrapping enabled (which is the default for multi-line edit controls), any wrapping that occurs as the user types will not produce linefeed characters (only Enter can do that). Whenever the user changes the control's content, the Change event is raised. TIP: To load a text file into an Edit control, use FileRead and GuiCtrl.Value. For example: MyEdit := MyGui.Add("Edit", "R20") MyEdit.Value := FileRead("C:\My File.txt") Edit Options To remove an option rather than adding it, precede it with a minus sign: Limit: Restricts the user's input to the visible width of the edit field. Alternatively, to limit input to a specific number of characters, include a number immediately afterward. For example, Limit10 would allow no more than 10 characters to be entered. Lowercase: The characters typed by the user are automatically converted to lowercase. Multi: Makes it possible to have more than one line of text. However, it is usually not necessary to specify this because it will be auto-detected based on height (H), rows (R), or contents (Text). Number: Prevents the user from typing anything other than digits into the field (however, it is still possible to paste non-digits into it). An alternate way of forcing a numeric entry is to attach an UpDown control to the Edit. Password: Hides the user's input (such as for password entry) by substituting masking characters for what the user types. If a non-default masking character is desired, include it immediately after the word Password. For example, Password* would make the masking character an asterisk rather than the black circle (bullet). Note: This option has no effect for multi-line edit controls. ReadOnly: Prevents the user from changing the control's contents. However, the text can still be scrolled, selected and copied to the clipboard. Tn: The letter T may be used to set tab stops inside a multi-line edit control (since tab stops determine the column positions to which literal TAB characters will jump, they can be used to format the text into columns). If the letter T is not used, tab stops are set at every 32 dialog units (the width of each "dialog unit" is determined by the operating system). If the letter T is used only once, tab stops are set at every n units across the entire width of the control. For example, MyGui.Add("Edit", "vMyEdit r16 t64") would double the default distance between tab stops. To have custom tab stops, specify the letter T multiple times as in the following example: MyGui.Add ("Edit", "vMyEdit r16 t8 t16 t32 t64 t128"). One tab stop is set for each of the absolute column positions in the list, up to a maximum of 50 tab stops. Note: Tab stops require a multiline edit control. Uppercase: The characters typed by the user are automatically converted to uppercase. WantCtrlA: Specify -WantCtrlA (minus WantCtrlA) to prevent the user's press of Ctrl+A from selecting all text in the edit control. WantReturn: Specify -WantReturn (minus WantReturn) to prevent a multi-line edit control from capturing Enter. Pressing Enter will then be the same as pressing the window's default button (if any). In this case, the user may press Ctrl+Enter to start a new line. WantTab: Causes Tab to produce a tab character rather than navigating to the next control. Without this option, the user may press Ctrl+Tab to produce a tab character inside a multi-line edit control. Note: WantTab also works in a single-line edit control. Wrap: Specify -Wrap (minus Wrap) to turn off word-wrapping in a multi-line edit control. Since this style cannot be changed after the control has been created, use one of the following to change it: 1) Destroy then recreate the window and its control; or 2) Create two overlapping edit controls, one with wrapping enabled and the other without it. The one not currently in use can be kept empty and/or hidden. See general options for other options like Right, Center, and Hidden. See also: position and sizing of controls. A more powerful edit control: HiEdit is a free, multitabbed, large-file edit control consuming very little memory. It can edit both text and binary files. For details and a demonstration, see HiEdit on GitHub. UpDown Description: A pair of arrow buttons that the user can click to increase or decrease a value. By default, an UpDown control automatically snaps onto the previously added control. This previous control is known as the UpDown's buddy control. The most common

example is a "spinner", which is an UpDown attached to an Edit control. For example: MyGui.Add("Edit") MyGui.Add("UpDown", "vMyUpDown Range1-10", 5) Or: MyGui.AddEdit() MyGui.AddUpDown("vMyUpDown Range1-10", 5) Appearance: In the example above, the Edit control is the UpDown's buddy control. Whenever the user presses one of the arrow buttons, the number in the Edit control is automatically increased or decreased. An UpDown's buddy control can also be a Text control or ListBox. However, due to OS limitations, controls other than these (such as ComboBox and DropDownList) might not work properly with the Change event and other features. Specify the UpDown's starting position as the last parameter (if omitted, it starts off at 0 or the number in the allowable range that is closest to 0). When MyGui.Submit or GuiCtrl.Value is used, the return value is the current numeric position of the UpDown. If the UpDown is attached to an Edit control and you do not wish to validate the user's input, it is best to use the UpDown's value rather than the Edit's. This is because the UpDown will always yield an in-range number, even when the user has typed something non-numeric or out-of-range in the Edit control. On a related note, numbers with more than three digits get a thousands separator (such as comma) by default. These separators are returned by the Edit control but not by the UpDown control. Whenever the user clicks one of the arrow buttons or presses an arrow key on the keyboard, the Change event is raised. UpDown Options Horz: Make's the control's buttons point left/right rather than up/down. By default, Horz also makes the control isolated (no buddy). This can be overridden by specifying Horz 16 in the control's options. Left: Puts the UpDown on the left side of its buddy rather than the right. Range: Sets the range to be something other than 0 to 100. After the word Range, specify the minimum, a dash, and maximum. For example, Range1-1000 would allow a number between 1 and 1000 to be selected; Range-50-50 would allow a number between -50 and 50; and Range-10--5 would allow a number between -10 and -5. The minimum and maximum may be swapped to cause the arrows to move in the opposite of their normal direction. The broadest allowable range is -2147483648-2147483647. Finally, if the buddy control is a ListBox, the range defaults to 32767-0 for verticals and the inverse for horizontals (Horz). Wrap: Causes the control to wrap around to the other end of its range when the user attempts to go beyond the minimum or maximum. Without Wrap, the control stops when the minimum or maximum is reached. 16: Specify -16 (minus 16) to cause a vertical UpDown to be isolated; that is, it will have no buddy. This also causes the control to obey any specified width, height, and position rather than conforming to the size of its buddy control. In addition, an isolated UpDown tracks its own position internally. This position can be retrieved normally by means such as MyGui.Submit. 0x80: Include 0x80 in Options to omit the thousands separator that is normally present between every three decimal digits in the buddy control. However, this style is normally not used because the separators are omitted from the number whenever the script retrieves it from the UpDown control itself (rather than its buddy control). Increments other than 1: In this script, NumEric demonstrates how to change an UpDown's increment to a value other than 1 (such as 5 or 0.1). Hexadecimal number format: The number format displayed inside the buddy control may be changed from decimal to hexadecimal by following this example: SendMessage 0x046D, 16, 0, "msctls_updown321" ; 0x046D is UDM_SETBASE However, this affects only the buddy control, not the UpDown's reported position. See also: position and sizing of controls. Picture (or Pic) Description: An area containing an image (see last two paragraphs for supported file types). The last parameter is the filename of the image, which is assumed to be in A_WorkingDir if an absolute path isn't specified. For example: MyGui.Add("Picture", "w300 h-1", "C:\My Pictures\Company Logo.gif") Or: MyGui.AddPicture("w300 h-1", "C:\My Pictures\Company Logo.gif") To retain the image's actual width and/or height, omit the W and/or H options. Otherwise, the image is scaled to the specified width and/or height (this width and height also determines which icon to load from a multi-icon .ICO file). To shrink or enlarge the image while preserving its aspect ratio, specify -1 for one of the dimensions and a positive number for the other. For example, specifying w200 h-1 would make the image 200 pixels wide and cause its height to be set automatically. If the picture cannot be loaded or displayed (e.g. file not found), an error is thrown and the control is not added. Picture controls support the Click and DoubleClick events, with the same caveat as Text controls. To use a picture as a background for other controls, the picture should normally be added prior to those controls. However, if those controls are input-capable and the picture has the SS_NOTIFY style (which may be added automatically by OnEvent), create the picture after the other controls and include 0x4000000 (which is WS_CLIPSIBLINGS) in the picture's Options. This trick also allows a picture to be the background behind a Tab control or ListView. Icons, cursors, and animated cursors: Icons and cursors may be loaded from the following types of files: ICO, CUR, ANI, EXE, DLL, CPL, SCR, and other types that contain icon resources. To use an icon group other than the first one in the file, include in Options the word Icon followed by the number of the group. In the following example, the default icon from the second icon group would be used: MyGui.Add("Picture", "Icon2", "C:\My Application.exe"). Specifying the word AltSubmit in Options tells the program to use Microsoft's GDIPlus.dll to load the image, which might result in a different appearance for GIF, BMP, and icon images. For example, it would load a GIF that has a transparent background as a transparent bitmap, which allows the BackgroundTrans option to take effect (but icons support transparency without AltSubmit). Formats supported without the use of GDIPlus include GIF, JPG, BMP, ICO, CUR, and ANI images. GDIPlus is used by default for other image formats, such as PNG, TIF, Exif, WMF and EMF. Animated GIFs: Although animated GIF files can be displayed in a picture control, they will not actually be animated. To solve this, use the AniGIF DLL (which is free for non-commercial use) as demonstrated at the AutoHotkey Forums. Alternatively, the ActiveX control type can be used. For example: ; Specify below the path to the GIF file to animate (local files are allowed too): pic :=

"http://www.animatedgif.net/cartoons/A_5odie_e0.gif" MyGui := Gui() MyGui.Add("ActiveX", "w100 h150", "mshtml: ") MyGui.Show A bitmap or icon handle can be used instead of a filename. For example, "HBITMAP:" handle. Button Description: A pushbutton, which can be pressed to trigger an action. In this case, the last parameter is the name of the button (shown on the button itself), which may include linefeeds (`n) to start new lines. For example: MyBtn := MyGui.Add("Button", "Default w80", "OK") MyBtn.OnEvent("Click", MyBtn_Click) ; Call MyBtn_Click when clicked. Or: MyBtn := MyGui.AddButton ("Default w80", "OK") MyBtn.OnEvent("Click", MyBtn_Click) ; Call MyBtn_Click when clicked. Appearance: The Click event is raised whenever the user clicks the button or presses Space or Enter while it has the focus. The DoubleClick, Focus and LoseFocus events are also supported. As these events are only raised if the control has the BS_NOTIFY (0x4000) style, it is added automatically by OnEvent. The example above includes the word Default in its Options to make "OK" the default button. The default button's Click event is automatically triggered whenever the user presses Enter, except when the keyboard focus is on a different button or a multi-line edit control having the WantReturn style. To later change the default button to another button, follow this example, which makes the Cancel button become the default: MyGui["Cancel"].Opt("+Default"). To later change the window to have no default button, follow this example: MyGui["OK"].Opt("-Default"). An ampersand (&) may be used in the button name to underline one of its letters. For example: MyGui.Add("Button",, "&Pause") In the example above, the letter P will be underlined, which allows the user to press Alt+P as shortcut key. To display a literal ampersand, specify two consecutive ampersands (&&). Known limitation: Certain desktop themes might not display a button's text properly. If this occurs, try including -Wrap (minus Wrap) in the control's options. However, this also prevents having more than one line of text. Checkbox Description: A small box that can be checked or unchecked to represent On/Off, Yes/No, etc. For example: MyGui.Add("CheckBox", "vShipToBillingAddress", "Ship to billing address?") Or: MyGui.AddCheckBox("vShipToBillingAddress", "Ship to billing address?") Appearance: The last parameter is a label displayed next to the box, which is typically used as a prompt or description of what the checkbox does. It may include linefeeds (`n) to start new lines. If a width (W) is specified in Options but no rows (R) or height (H), the control's text will be word-wrapped as needed, and the control's height will be set automatically. GuiCtrl.Value returns the number 1 for checked, 0 for unchecked, and -1 for gray/indeterminate. Specify the word Check3 in Options to enable a third "indeterminate" state that displays a gray checkmark or a square instead of a black checkmark (the indeterminate state indicates that the checkbox is neither checked nor unchecked). Specify the word Checked or CheckedGray in Options to have the checkbox start off checked or indeterminate, respectively. The word Checked may optionally be followed immediately by a 0, 1, or -1 to indicate the starting state. In other words, "Checked" and "Checked" VarContainingOne are the same. Whenever the checkbox is clicked, it automatically cycles between its two or three possible states, and then raises the Click event, allowing the script to immediately respond to the user's input. The DoubleClick, Focus and LoseFocus events are also supported. As these events are only raised if the control has the BS_NOTIFY (0x4000) style, it is added automatically by OnEvent. This style is not applied by default as it prevents rapid clicks from changing the state of the checkmark (such as if the user clicks twice to toggle from unchecked to checked and then to indeterminate). Known limitation: Certain desktop themes might not display a checkbox's text properly. If this occurs, try including -Wrap (minus Wrap) in the control's options. However, this also prevents having more than one line of text. Radio Description: A radio button is a small empty circle that can be checked (on) or unchecked (off). For example: MyGui.Add("Radio", "vMyRadioGroup", "Wait for all items to be in stock before shipping.") Or: MyGui.AddRadio("vMyRadioGroup", "Wait for all items to be in stock before shipping.") Appearance: These controls usually appear in radio groups, each of which contains two or more radio buttons. When the user clicks a radio button to turn it on, any others in its radio group are turned off automatically (the user may also navigate inside a group with the arrow keys). A radio group is created automatically around all consecutively added radio buttons. To start a new group, specify the word Group in the Options of the first button of the new group -- or simply add a non-radio control in between, since that automatically starts a new group. For the last parameter, specify the label to display to the right of the radio button. This label is typically used as a prompt or description, and it may include linefeeds (`n) to start new lines. If a width (W) is specified in Options but no rows (R) or height (H), the control's text will be word-wrapped as needed, and the control's height will be set automatically. Specify the word Checked in Options to have the button start off in the "on" state. The word Checked may optionally be followed immediately by a 0 or 1 to indicate the starting state: 0 for unchecked and 1 for checked. In other words, "Checked" and "Checked" VarContainingOne are the same. GuiCtrl.Value returns the number 1 for "on" and 0 for "off". To instead retrieve the position number of the selected radio option within a radio group, name only one of the radio buttons and use MyGui.Submit. The Click event is raised whenever the user turns on the button. Unlike the single-variable mode in the previous paragraph, the event callback must be registered for each button in a radio group for which it should be called. This allows the flexibility to ignore the clicks of certain buttons. The DoubleClick, Focus and LoseFocus events are also supported. As these events are only raised if the control has the BS_NOTIFY (0x4000) style, it is added automatically by OnEvent. Known limitation: Certain desktop themes might not display a radio button's text properly. If this occurs, try including -Wrap (minus Wrap) in the control's options. However, this also

prevents having more than one line of text. DropDownList (or DDL) Description: A list of choices that is displayed in response to pressing a small button. In this case, the last parameter of MyGui.Add is an Array like ["Choice1","Choice2","Choice3"]. For example: MyGui.Add("DropDownList", "vColorChoice", ["Black","White","Red"]) Or: MyGui.AddDropDownList("vColorChoice", ["Black","White","Red"]) Appearance: To have one of the items pre-selected when the window first appears, include in Options the word Choose followed immediately by the number of an item to be pre-selected. For example, Choose5 would pre-select the fifth item (as with other options, it can also be a variable such as "Choose" Var). After the control is created, use GuiCtrl.Value, GuiCtrl.Text or GuiCtrl.Choose to change the selection, and GuiCtrl.Add or GuiCtrl.Delete to add or remove entries from the list. Specify either the word Uppercase or Lowercase in Options to automatically convert all items in the list to uppercase or lowercase. Specify the word Sort to automatically sort the contents of the list alphabetically (this also affects any items added later via GuiCtrl.Add). The Sort option also enables incremental searching whenever the list is dropped down; this allows an item to be selected by typing the first few characters of its name. When MyGui.Submit or GuiCtrl.Value is used, the return value is the position number of the currently selected item (the first item is 1, the second is 2, etc.) or zero if none is selected. To get its text instead, use GuiCtrl.Text. Whenever the user selects a new item, the Change event is raised. The Focus and LoseFocus events are also supported. Use the R or H option to control the height of the popup list. For example, specifying R5 would make the list 5 rows tall, while H400 would set the total height of the selection field and list to 400 pixels. If both R and H are omitted, the list will automatically expand to take advantage of the available height of the user's desktop. To set the height of the selection field or list items, use the CB_SETITEMHEIGHT message as in the example below: MyGui := Gui() DDL := MyGui.Add("DDL", "vcbx w200 Choose1", ["One","Two"]) ; CB_SETITEMHEIGHT = 0x0153 PostMessage(0x0153, -1, 50, DDL) ; Set height of selection field. PostMessage(0x0153, 0, 50, DDL) ; Set height of list items. MyGui.Show("h70") ComboBox Description: Same as DropDownList but also permits free-form text to be entered as an alternative to picking an item from the list. For example: MyGui.Add("ComboBox", "vColorChoice", ["Red","Green","Blue"]) Or: MyGui.AddComboBox("vColorChoice", ["Red","Green","Blue"]) Appearance: In addition to allowing all the same options as DropDownList above, the word Limit may be included in Options to restrict the user's input to the visible width of the ComboBox's edit field. Also, the word Simple may be specified to make the ComboBox behave as though it is an Edit field with a ListBox beneath it. GuiCtrl.Value returns the position number of the currently selected item (the first item is 1, the second is 2, etc.) or 0 if the control contains text which does not match a list item. To get the contents of the ComboBox's edit field, use GuiCtrl.Text. MyGui.Submit stores the text, unless the word AltSubmit is in the control's Options and the text matches a list item, in which case it stores the position number of the item. Whenever the user selects a new item or changes the control's text, the Change event is raised. The Focus and LoseFocus events are also supported. ListBox Description: A relatively tall box containing a list of choices that can be selected. In this case, the last parameter of MyGui.Add is an Array like ["Choice1","Choice2","Choice3"]. For example: MyGui.Add("ListBox", "r5 vColorChoice", ["Red","Green","Blue"]) Or: MyGui.AddListBox("r5 vColorChoice", ["Red","Green","Blue"]) Appearance: To have one of the items pre-selected when the window first appears, include in Options the word Choose followed immediately by the number of an item to be pre-selected. For example, Choose5 would pre-select the fifth item. To have multiple items pre-selected, use GuiCtrl.Choose multiple times (requires the Multi option). After the control is created, use GuiCtrl.Value, GuiCtrl.Text or GuiCtrl.Choose to change the selection, and GuiCtrl.Add or GuiCtrl.Delete to add or remove entries from the list. If the Multi option is absent, GuiCtrl.Value returns the position number of the currently selected item (the first item is 1, the second is 2, etc.) or 0 if there is no item selected. To get the selected item's text instead, use GuiCtrl.Text. If the Multi option is used, GuiCtrl.Value and GuiCtrl.Text return an array of items instead of a single item. MyGui.Submit stores GuiCtrl.Text by default, but stores GuiCtrl.Value instead if the word AltSubmit is in the control's Options. Whenever the user selects or deselects one or more items, the Change event is raised. The DoubleClick, Focus and LoseFocus events are also supported. When adding a large number of items to a ListBox, performance may be improved by using MyListBox.Opt("-Redraw") prior to the operation, and MyListBox.Opt("+Redraw") afterward. See Redraw for more details. ListBox Options Choose: See above. Multi: Allows more than one item to be selected simultaneously via shift-click and control-click (to avoid the need for shift/control-click, specify the number 8 instead of the word Multi). In this case, MyGui.Submit or GuiCtrl.Value returns an array of selected position numbers. For example, [1, 2, 3] would indicate that the first three items are selected. To get an array of selected texts instead, use GuiCtrl.Text. To extract the individual items from the array, use MyListBox.Text[1] (1 would be the first item) or a For-loop such as this example: For Index, Field in MyListBox.Text { MsgBox "Selection number " Index " is " Field } ReadOnly: Prevents items from being visibly highlighted when they are selected (but MyGui.Submit, GuiCtrl.Value or GuiCtrl.Text will still return the selected item). Sort: Automatically sorts the contents of the list alphabetically (this also affects any items added later via GuiCtrl.Add). The Sort option also enables incremental searching, which allows an item to be selected by typing the first few characters of its name. Tn: The letter T may be used to set tab stops, which can be used to format the text into columns. If the letter T is not used, tab stops are set at every 32 dialog units (the width of each "dialog unit" is determined by the operating system). If the letter T is used only once, tab stops are set at every n units across the entire width of the control. For example, MyGui.Add("ListBox", "vMyListBox t64") would double the default distance between tab stops. To have custom tab stops, specify the letter T multiple times as in the following example: MyGui.Add("ListBox", "vMyListBox t8 t16 t32 t64 t128"). One tab stop is set for each of the absolute column positions in the list, up to a maximum of 50 tab stops. 0x100: Include 0x100 in Options to turn on the LBS_NOINTEGRALHEIGHT style. This forces the ListBox to be exactly the height specified rather than a height that prevents a partial row from appearing at the bottom. This option also prevents the ListBox from shrinking when its font is changed. To specify the number of rows of text (or the height and width), see position and sizing of controls. ListView and TreeView See separate pages ListView and TreeView. Link Description: A text control that can contain links similar to those found in a web browser. Within the control's text, enclose the link text within and to create a clickable link. Although this looks like HTML, Link controls only support the opening tag (optionally with an ID and/or HREF attribute) and closing tag. For example: MyGui.Add("Link",, 'This is a link') MyGui.Add("Link",, 'Links may be used anywhere in the text like this or that') Or: MyGui.AddLink(, 'This is a link') MyGui.AddLink(, 'Links may be used anywhere in the text like this or that') Appearance: Whenever the user clicks on a link, the Click event is raised. If the control has no Click callback (registered by calling OnEvent), the link's HREF is automatically executed as though passed to the Run function. MyGui := Gui() Link := MyGui.Add("Link", , 'Click to run Notepad or open ' . 'online help.') Link.OnEvent("Click", Link_Click) Link_Click(Ctrl, ID, HREF) { if MsgBox("ID: " ID "`nHREF: " HREF "`n`nExecute this link?",, "y/n") = "yes" Run(HREF) } MyGui.Show() Hotkey Description: A box that looks like a single-line edit control but instead accepts a keyboard combination pressed by the user. For example, if the user presses Ctrl+Alt+C on an English keyboard layout, the box would display "Ctrl + Alt + C". For example: MyGui.Add("Hotkey", "vChosenHotkey") Or: MyGui.AddHotkey("vChosenHotkey") Appearance: GuiCtrl.Value returns the control's hotkey modifiers and name, which are compatible with the Hotkey function. Examples: ^!C, +!Home, +^Down, ^Numpad1, ! NumpadEnd. If there is no hotkey in the control, the value is blank. Note: Some keys are displayed the same even though they are retrieved as different names. For example, both ^Numpad7 and ^NumpadHome might be displayed as Ctrl + Num 7. By default, the control starts off with no hotkey specified. To instead have a default, specify its modifiers and name as the last parameter as in this example: MyGui.Add("Hotkey", "vChosenHotkey", "^!p"). The only modifiers supported are ^ (Ctrl), ! (Alt), and + (Shift). See the key list for available key names. Whenever the user changes the control's content (by pressing a key), the Change event is raised. Note: The event is raised even when an incomplete hotkey is present. For example, if the user holds down Ctrl, the event is raised once and Value returns only a circumflex (^). When the user completes the hotkey, the event is raised again and Value returns the complete hotkey. To restrict the types of hotkeys the user may enter, include the word Limit followed by the sum of one or more of the following numbers: 1: Prevent unmodified keys 2: Prevent Shift-only keys 4: Prevent Ctrl-only keys 8: Prevent Alt-only keys 16: Prevent Shift+Ctrl keys 32: Prevent Shift+Alt keys 64: This value is not supported (it will not behave correctly) 128: Prevent Shift+Ctrl+Alt keys For example, Limit1 would prevent unmodified hotkeys such as letters and numbers from being entered, and Limit15 would require at least two modifier keys. If the user types a forbidden modifier combination, the Ctrl+Alt combination is automatically and visibly substituted. The Hotkey control has limited capabilities. For example, it does not support mouse/joystick hotkeys or Win (LWin and RWin). One way to work around this is to provide one or more checkboxes as a means for the user to enable extra modifiers such as Win. DateTime Description: A box that looks like a single-line edit control but instead accepts a date and/or time. A drop-down calendar is also provided. For example: MyGui.Add("DateTime", "vMyDateTime", "LongDate") Or: MyGui.AddDateTime ("vMyDateTime", "LongDate") Appearance: The last parameter is a format string, as described below. SetFormat Sets the display format of a DateTime control. DateTime.SetFormat(Format) Format Type: String One of the following: ShortDate (or omitted/blank): Uses the locale's short date format. For example, in some locales it would look like: 6/1/2005 LongDate: Uses the locale's long date format. For example, in some locales it would look like: Wednesday, June 01, 2005 Time: Shows only the time using the locale's time format. Although the date is not shown, it is still present in the control and will be retrieved along with the time in the YYYYMMDDHH24MISS format. For example, in some locales it would look like: 9:37:45 PM (custom format): Specify any combination of date and time formats. For example, "M/d/yy HH:mm" would look like 6/1/05 21:37. Similarly, "dddd MMMM d, yyyy hh:mm:ss tt" would look like Wednesday June 1, 2005 09:37:45 PM. Letters and numbers to be displayed literally should be enclosed in single quotes as in this example: "'Date:' MM/dd/yy 'Time:' hh:mm:ss tt". By contrast, non-alphanumeric characters such as spaces, tabs, slashes, colons, commas, and other punctuation do not need to be enclosed in single quotes. The exception to this is the single quote character itself: to produce it literally, use four consecutive single quotes (''''), or just two if the quote is already inside an outer pair of quotes. DateTime Usage To have a date other than today pre-selected, include in Options the word Choose followed immediately by a date in YYYYMMDD format. For example, Choose20050531 would pre-select May 31, 2005 (as with other options, it can also be a variable such as "Choose" Var). To have no date/time selected, specify ChooseNone. ChooseNone also creates a checkbox inside the control that is unchecked whenever the control has no date. Whenever the control has no date, MyGui.Submit or GuiCtrl.Value will retrieve a blank value (empty string). The time of day may optionally be present. However, it must always be preceded by a date when going into or coming out of the control. The format of the time portion is HH24MISS (hours, minutes, seconds), where HH24 is expressed in 24-hour format; for example, 09 is 9am and 21 is 9pm. Thus, a complete date-time string would have the format YYYYMMDDHH24MISS. When specifying dates in the

YYYYMMDDHH24MISS format, only the leading part needs to be present. Any remaining element that has been omitted will be supplied with the following default values: MM with month 01, DD with day 01, HH24 with hour 00, MI with minute 00 and SS with second 00. Within the drop-down calendar, the today-string at the bottom can be clicked to select today's date. In addition, the year and month name are clickable and allow easy navigation to a new month or year. Keyboard navigation: Use the ?/? arrow keys, the +/- numpad keys, and Home/End to increase or decrease the control's values. Use ? and ? to move from field to field inside the control. Within the drop-down calendar, use the arrow keys to move from day to day; use PgUp/PgDn to move backward/forward by one month; and use Home/End to select the first/last day of the month. When MyGui.Submit or GuiCtrl.Value is used, the return value is the selected date and time in YYYYMMDDHH24MISS format. Both the date and the time are present regardless of whether they were actually visible in the control. Whenever the user changes the date or time, the Change event is raised. The Focus and LoseFocus events are also supported. DateTime Options Choose: See above. Range: Restricts how far back or forward in time the selected date can be. After the word Range, specify the minimum and maximum dates in YYYYMMDD format (with a dash between them). For example, Range20050101-20050615 would restrict the date to the first 5.5 months of 2005. Either the minimum or maximum may be omitted to leave the control unrestricted in that direction. For example, Range20010101 would prevent a date prior to 2001 from being selected and Range-20091231 (leading dash) would prevent a date later than 2009 from being selected. Without the Range option, any date between the years 1601 and 9999 can be selected. The time of day cannot be restricted. Right: Causes the drop-down calendar to drop down on the right side of the control instead of the left. 1: Specify the number 1 in Options to provide an up-down control to the right of the control to modify date-time values, which replaces the button of the drop-down month calendar that would otherwise be available. This does not work in conjunction with the format option LongDate described above. 2: Specify the number 2 in Options to provide a checkbox inside the control that the user may uncheck to indicate that no date/time is selected. Once the control is created, this option cannot be changed. Colors inside the drop-down calendar: The colors of the day numbers inside the drop-down calendar obey that set by Gui.SetFont or the c (Color) option. To change the colors of other parts of the calendar, follow this example: SendMessage 0x1006, 4, 0xFFAA99, "SysDateTimePick321" ; 0x1006 is DTM_SETMCCOLOR. 4 is MCSC_MONTHBK (background color). The color must be specified in BGR vs. RGB format (red and blue components swapped). MonthCal Description: A tall and wide control that displays all the days of the month in calendar format. The user may select a single date or a range of dates. For example: MyGui.Add("MonthCal", "vMyCalendar") Or: MyGui.AddMonthCal("vMyCalendar") Appearance: To have a date other than today pre-selected, specify it as the third parameter in YYYYMMDD format (e.g. 20050531). A range of dates may also be pre-selected by including a dash between two dates (e.g. "20050525-20050531"). It is usually best to omit width (W) and height (H) for a MonthCal because it automatically sizes itself to fit exactly one month. To display more than one month vertically, specify R2 or higher in Options. To display more than one month horizontally, specify W-2 (W negative two) or higher. These options may both be present to expand in both directions. The today-string at the bottom of the control can be clicked to select today's date. In addition, the year and month name are clickable and allow easy selection of a new year or month. Keyboard navigation: Keyboard navigation is fully supported in MonthCal, but only if it has the keyboard focus. For supported keyboard shortcuts, see DateTime's keyboard navigation (within the drop-down calendar). When MyGui.Submit or GuiCtrl.Value is used, the return value is the selected date in YYYYMMDD format (without any time portion). However, when the multi-select option is in effect, the minimum and maximum dates are retrieved with a dash between them (e.g. 20050101-20050108). If only a single date was selected in a multi-select calendar, the minimum and maximum are both present but identical. StrSplit can be used to separate the dates. For example, the following would put the minimum in Date[1] and the maximum in Date[2]: Date := StrSplit(MyMonthCal.Value, "-"). The Change event is raised when the user changes the selection. When specifying dates in the YYYYMMDD format, the MM and/or DD portions may be omitted, in which case they are assumed to be 1. For example, 200205 is seen as 20020501, and 2005 is seen as 20050101. MonthCal Options Multi: Multi-select. Allows the user to shift-click or click-drag to select a range of adjacent dates (the user may still select a single date too). This option may be specified explicitly or put into effect automatically by means of specifying a selection range when the control is created. For example: MyGui.Add("MonthCal", "vMyCal", "20050101-20050108"). Once the control is created, this option cannot be changed. Range: Restricts how far back or forward in time the calendar can go. After the word Range, specify the minimum and maximum dates in YYYYMMDD format (with a dash between them). For example, Range20050101-20050615 would restrict the selection to the first 5.5 months of 2005. Either the minimum or maximum may be omitted to leave the calendar unrestricted in that direction. For example, Range20010101 would prevent a date prior to 2001 from being selected and Range-20091231 (leading dash) would prevent a date later than 2009 from being selected. Without the Range option, any date between the years 1601 and 9999 can be selected. 4: Specify the number 4 in Options to display week numbers (1-52) to the left of each row of days. Week 1 is defined as the first week that contains at least four days. 8: Specify the number 8 in Options to prevent the circling of today's date within the control. 16: Specify the number 16 in Options to prevent the display of today's date at the bottom of the control. Colors: The colors of the day numbers inside the calendar obey that set by Gui.SetFont or the c (Color) option. To change the colors of other parts of the calendar, follow this example: SendMessage 0x100A, 5, 0xFFAA99, "SysMonthCal321" ; 0x100A is MCM_SETCOLOR. 5 is MCSC_TITLETEXT (color of title text). The color must be specified in BGR vs. RGB format (red and blue components swapped). Slider Description: A sliding bar that the user can move along a vertical or horizontal track. The standard volume control in the taskbar's tray is an example of a slider. For example: MyGui.Add("Slider", "vMySlider", 50) Or: MyGui.AddSlider("vMySlider", 50) Appearance: Specify the starting position of the slider as the last parameter. If the last parameter is omitted, the slider starts off at 0 or the number in the allowable range that is closest to 0. The user may slide the control by the following means: 1) dragging the bar with the mouse; 2) clicking inside the bar's track area with the mouse; 3) turning the mouse wheel while the control has focus; or 4) pressing the following keys while the control has focus: ?, ?, ?, ?, PgUp, PgDn, Home, and End. GuiCtrl.Value and MyGui.Submit return or store the current numeric position of the slider. Detecting Changes By default, the slider's Change event is raised when the user has stopped moving the slider, such as by releasing the mouse button after having dragging it. If the control has the AltSubmit option, the Change event is also raised (very frequently) after each visible movement of the bar while the user is dragging it with the mouse. Ctrl_Change(GuiCtrlObj, Info) Info Type: Integer A numeric value from the tables below indicating how the slider was moved. These values and the corresponding names are defined in the Windows SDK. ValueNameMeaning 0TB_LINEUPThe user pressed ? or ?. 1TB_LINEDOWNThe user pressed ? or ?. 2TB_PAGEUPThe user pressed PgUp. 3TB_PAGEDOWNThe user pressed PgDn. 4TB_THUMBPOSITIONThe user moved the slider via the mouse wheel, or finished a drag-and-drop to a new position. 6TB_TOPThe user pressed Home to send the slider to the left or top side. 7TB_BOTTOMThe user pressed End to send the slider to the right or bottom side. Only if the AltSubmit option is used: ValueNameMeaning 5TB_THUMBTRACKThe user is currently dragging the slider via the mouse; that is, the mouse button is currently down. 8TB_ENDTRACKThe user has finished moving the slider, either via the mouse or the keyboard. Note: With the exception of mouse wheel movement (#4), the Change event is raised again for #8 even though it was already raised with one of the digits above. Slider Options Buddy1 and Buddy2: Specifies up to two existing controls to automatically reposition at the ends of the slider. Buddy1 is displayed at the left or top side (depending on whether the Vertical option is present). Buddy2 is displayed at the right or bottom side. After the word Buddy1 or Buddy2, specify the Name or HWND of an existing control. For example, Buddy1MyTopText would assign the control whose name is MyTopText. The text or ClassNN of a control can also be used, but only up to the first space or tab. Center: The thumb (the bar moved by the user) will be blunt on both ends rather than pointed at one end. Invert: Reverses the control so that the lower value is considered to be on the right/bottom rather than the left/top. This is typically used to make a vertical slider move in the direction of a traditional volume control. Note: The ToolTip option described below will not obey the inversion and therefore should not be used in this case. Left: The thumb (the bar moved by the user) will point to the top rather than the bottom. But if the Vertical option is in effect, the thumb will point to the left rather than the right. Line: Specifies the number of positions to move when the user presses one of the arrow keys. After the word Line, specify number of positions to move. For example: Line2. NoTicks: Omits tickmarks alongside the track. Page: Specifies the number of positions to move when the user presses PgUp or PgDn. After the word Page, specify number of positions to move. For example: Page10. Range: Sets the range to be something other than 0 to 100. After the word Range, specify the minimum, a dash, and maximum. For example, Range1-1000 would allow a number between 1 and 1000 to be selected; Range-50-50 would allow a number between -50 and 50; and Range-10--5 would allow a number between -10 and -5. Thick: Specifies the length of the thumb (the bar moved by the user). After the word Thick, specify the thickness in pixels (e.g. Thick30). To go beyond a certain thickness, it is probably necessary to either specify the Center option or remove the theme from the control (which can be done by specifying -Theme in the control's options). TickInterval: Provides tickmarks alongside the track at the specified interval. After the word TickInterval, specify the interval at which to display additional tickmarks (if the interval is never set, it defaults to 1). For example, TickInterval10 would display a tickmark once every 10 positions. ToolTip: Creates a tooltip that reports the numeric position of the slider as the user is dragging it. To have the tooltip appear in a non-default position, specify one of the following instead: ToolTipLeft or ToolTipRight (for vertical sliders); ToolTipTop or ToolTipBottom (for horizontal sliders). Vertical: Makes the control slide up and down rather than left and right. The above options can be changed via GuiCtrl.Opt after the control is created. Progress Description: A dual-color bar typically used to indicate how much progress has been made toward the completion of an operation. For example: MyGui.Add ("Progress", "w200 h20 cBlue vMyProgress", 75) Or: MyGui.AddProgress("w200 h20 cBlue vMyProgress", 75) Appearance: Specify the starting position of the bar as the third parameter (if omitted, the bar starts off at 0 or the number in the allowable range that is closest to 0). To later change the position of the bar, follow these examples, all of which operate upon a progress bar whose Name is MyProgress: MyGui["MyProgress"].Value += 20 ; Increase the current position by 20. MyGui["MyProgress"].Value := 50 ; Set the current position to 50. For horizontal Progress Bars, the thickness of the bar is equal to the control's height. For vertical Progress Bars it is equal to the control's width. Progress Options Cn: Changes the bar's color. Specify for n one of the 16 primary HTML color names or a 6-digit RGB color value. Examples: cRed, cFFFF33, cDefault. If the C option is never used (or cDefault is specified), the system's default bar color will be used. BackgroundN: Changes the bar's background color. Specify for n one of the 16 primary HTML color names or a 6-digit RGB color value. Examples: BackgroundGreen, BackgroundFFFF33, BackgroundDefault. If the Background option is never used (or BackgroundDefault is specified), the background color

will be that of the window or tab control behind it. Range: Sets the range to be something other than 0 to 100. After the word Range, specify the minimum, a dash, and maximum. For example, Range0-1000 would allow numbers between 0 and 1000; Range-50-50 would allow numbers between -50 and 50; and Range-10--5 would allow numbers between -10 and -5. Smooth: Displays a simple continuous bar. If this option is not used and the bar does not have any custom colors, the bar's appearance is defined by the current system theme. Otherwise, the bar appears as a length of segments. Vertical: Makes the bar rise or fall vertically rather than move along horizontally. The above options can be changed via GuiCtrl.Opt after the control is created. GroupBox Description: A rectangular border/frame, often used around other controls to indicate they are related. In this case, the last parameter is the title of the box, which if present is displayed at its upper-left edge. For example: MyGui.Add("GroupBox", "w200 h100", "Geographic Criteria") Or: MyGui.AddGroupBox("w200 h100", "Geographic Criteria") Appearance: By default, a GroupBox's title may have only one line of text. This can be overridden by specifying Wrap in Options. To specify the number of rows inside the control (or its height and width), see position and sizing of controls. Tab3 Description: A large control containing multiple pages, each of which contains other controls. From this point forward, these pages are referred to as "tabs". There are three types of Tab control: Tab3: Fixes some issues which affect Tab2 and Tab. Controls are placed within an invisible "tab dialog" which moves and resizes with the tab control. The tab control is themed by default. Tab2: Fixes rare redrawing problems in the original "Tab" control but introduces some other problems. Tab: Retained for backward compatibility because of differences in behavior between Tab2/Tab3 and Tab. For example: MyGui.Add("Tab3",, ["General","View","Settings"]) Or: MyGui.AddTab3(, ["General","View","Settings"]) Appearance: The last parameter above is an Array of tab names. To have one of the tabs pre-selected when the window first appears, include in Options the word Choose followed immediately by the number of a tab to be pre-selected. For example, Choose5 would pre-select the fifth tab (as with other options, it can also be a variable such as "Choose" Var). After the control is created, use Value, Text or Choose to change the selected tab, and Add or Delete to add or remove tabs. After creating a Tab control, subsequently added controls automatically belong to its first tab. This can be changed at any time by following these examples (in this case, Tab is the GuiControl object of the first tab control and Tab2 of the second one): Tab.UseTab() ; Future controls are not part of any tab control. Tab.UseTab(3) ; Future controls are owned by the third tab of the current tab control. Tab2.UseTab(3) ; Future controls are owned by the third tab of the second tab control. Tab.UseTab ("Name") ; Future controls are owned by the tab whose name starts with Name (not case sensitive). Tab.UseTab("Name", true) ; Same as above but requires exact match (not case sensitive). It is also possible to use any of the examples above to assign controls to a tab or tab-control that does not yet exist (except in the case of the Name method). But in that case, the relative positioning options described below are not supported. Positioning: When each tab of a Tab control receives its first sub-control, that sub-control will have a special default position under the following conditions: 1) The X and Y coordinates are both omitted, in which case the first sub-control is positioned at the upper-left corner of the tab control's interior (with a standard margin), and sub-controls beyond the first are positioned beneath the previous control; 2) The X+n and/or Y+n positioning options are specified, in which case the sub-control is positioned relative to the upper-left corner of the tab control's interior. For example, specifying x+10 y+10 would position the control 10 pixels right and 10 pixels down from the upper left corner. Current tab: GuiCtrl.Value returns the position number of the currently selected tab (the first tab is 1, the second is 2, etc.). To get its text instead, use GuiCtrl.Text. MyGui.Submit stores GuiCtrl.Text by default, but stores GuiCtrl.Value instead if the word AltSubmit is in the control's Options. Detecting tab selection: Whenever the user switches tabs, the Change event is raised. Keyboard navigation: The user may press Ctrl+PgDn/PgUp to navigate from page to page in a tab control; if the keyboard focus is on a control that does not belong to a Tab control, the window's first Tab control will be navigated. Ctrl+Tab and Ctrl+Shift+Tab may also be used except that they will not work if the currently focused control is a multi-line Edit control. Limits: Each window may have no more than 255 tab controls. Each tab control may have no more than 256 tabs (pages). In addition, a tab control may not contain other tab controls. Tab3 vs. Tab2 vs. Tab Parent window: The parent window of a control affects the positioning and visibility of the control and tab-key navigation order. If a sub-control is added to an existing Tab3 control, its parent window is the "tab dialog", which fills the tab control's display area. Most other controls, including sub-controls of Tab or Tab2 controls, have no parent other than the GUI window itself. Positioning: For Tab and Tab2, sub-controls do not necessarily need to exist within their tab control's boundaries: they will still be hidden and shown whenever their tab is selected or de-selected. This behavior is especially appropriate for the "buttons" style described below. For Tab3, sub-controls assigned to a tab before the tab control is created behave as though added to a Tab or Tab2 control. All other sub-controls are visible only within the display area of the tab control. If a Tab3 control is moved, its sub-controls are moved with it. Tab and Tab2 controls do not have this behavior. In the rare case that WinMove (or an equivalent DllCall) is used to move a control, the coordinates must be relative to the parent window of the control, which might not be the GUI (see above). By contrast, GuiCtrl.Move takes GUI coordinates and ControlMove takes window coordinates, regardless of the control's parent window. Autosizing: If not specified by the script, the width and/or height of the Tab3 control are automatically calculated at one of the following times (whichever comes first after the control is created): The first time the Tab3 control ceases to be the current tab control. This can occur as a result of calling GuiCtrl.UseTab (with or without parameters) or creating another tab control. The first time MyGui.Show is called for that particular Gui. The calculated size accounts for sub-controls which exist when autosizing occurs, plus the default margins. The size is calculated only once, and will not be recalculated even if controls are added later. If the Tab3 control is empty, it receives the same default size as a Tab or Tab2 control. Tab and Tab2 controls are not autosized; they receive an arbitrary default size. Tab-key navigation order: The navigation order via Tab usually depends on the order in which the controls are created. When tab controls are used, the order also depends on the type of tab control: Tab and Tab2 allow their sub-controls to be mixed with other controls within the tab-key order. Tab2 puts its tab buttons after its sub-controls in the tab-key order. Tab3 groups its sub-controls within the tab-key order and puts them after its tab buttons. Notification messages (Tab3): Common and Custom controls typically send notification messages to their parent window. Any WM_COMMAND, WM_NOTIFY, WM_VSCROLL, WM_HSCROLL or WM_CTLCOLOR' messages received by a Tab3 control's tab dialog are forwarded to the GUI window and can be detected by using OnMessage. If the tab control is themed and the sub-control lacks the +BackgroundTrans option, WM_CTLCOLORSTATIC is fully handled by the tab dialog and not forwarded. Other notification messages (such as custom messages) are not supported. Known issues with Tab2: BackgroundTrans has no effect inside a Tab2 control. WebBrowser controls do not redraw correctly. AnimateWindow and possibly other Win32 API calls can cause the tab's controls to disappear. Known issues with Tab: Activating a GUI window by clicking certain parts of its controls, such as scrollbars, might redraw improperly. BackgroundTrans has no effect if the Tab control contains a ListView. WebBrowser controls are invisible. Tab Options Choose: See above. Background: Specify -Background (minus Background) to override the window's custom background color and use the system's default Tab control color. Specify +Theme -Background to make the Tab control conform to the current desktop theme. However, most control types will look strange inside such a Tab control because their backgrounds will not match that of the tab control. This can be fixed for some control types (such as Text) by adding BackgroundTrans to their options. Buttons: Creates a series of buttons at the top of the control rather than a series of tabs (in this case, there will be no border by default because the display area does not typically contain controls). Left/Right/Bottom: Specify one of these words to have the tabs on the left, right, or bottom side instead of the top. See TCS_VERTICAL for limitations on Left and Right. Wrap: Specify -Wrap (minus Wrap) to prevent the tabs from taking up more than a single row (in which case if there are too many tabs to fit, arrow buttons are displayed to allow the user to slide more tabs into view). To specify the number of rows of text inside the control (or its height and width), see position and sizing of controls. Icons in Tabs: An icon may be displayed next to each tab's name/text via SendMessage. This is demonstrated in the forum topic Icons in tabs. StatusBar Description: A row of text and/or icons attached to the bottom of a window, which is typically used to report changing conditions. For example: SB := MyGui.Add("StatusBar",, "Bar's starting text (omit to start off empty).") SB.SetText("There are " . RowCount . " rows selected.") Or: SB := MyGui.AddStatusBar(, "Bar's starting text (omit to start off empty).") SB.SetText ("There are " . RowCount . " rows selected.") Appearance: The simplest use of a status bar is to call SB.SetText whenever something changes that should be reported to the user. To report more than one piece of information, divide the bar into sections via SB.SetParts. To display icon(s) in the bar, call SB.SetIcon. StatusBar Methods SetText Displays NewText in the specified part of the status bar. Success := SB.SetText(NewText , PartNumber, Style) NewText Type: String Up to two tab characters (`t) may be present anywhere in NewText: anything to the right of the first tab is centered within the part, and anything to the right of the second tab is right-justified.PartNumber Type: Integer If PartNumber is omitted, it defaults to 1. Otherwise, specify an integer between 1 and 256.Style Type: Integer If Style is omitted, it defaults to 0, which uses a traditional border that makes that part of the bar look sunken. Otherwise, specify 1 to have no border or 2 to have border that makes that part of the bar look raised. SetParts Divides the bar into multiple sections according to the specified widths (in pixels), and returns a non-zero value (the status bar's HWND). Hwnd := SB.SetParts(Width1, Width2, ... Width255) Width1 ... Width255 Type: Integer If all parameters are omitted, the bar is restored to having only a single, long part. Otherwise, specify the width of each part except the last (the last will fill the remaining width of the bar). For example, SB.SetParts(50, 50) would create three parts: the first two of width 50 and the last one of all the remaining width. Note: Any parts "deleted" by SB.SetParts() will start off with no text the next time they are shown (furthermore, their icons are automatically destroyed). SetIcon Displays a small icon to the left of the text in the specified part, and returns the icon's handle. HICON := SB.SetIcon(Filename , IconNumber, PartNumber) FileName Type: String The path of an icon or image file. For a list of supported formats, see the Picture control. A bitmap or icon handle can be used instead of a filename. For example, SB.SetIcon ("HICON:" handle). IconNumber Type: Integer To use an icon group other than the first one in the file, specify its number for IconNumber. For example, SB.SetIcon("Shell32.dll", 2) would use the default icon from the second icon group. If IconNumber is negative, its absolute value is assumed to be the resource ID of an icon within an executable file.PartNumber Type: Integer If PartNumber is omitted, it defaults to 1. Otherwise, specify an integer between 1 and 256. Note: The HICON is a system resource that can be safely ignored by most scripts because it is destroyed automatically when the status bar's window is destroyed. Similarly, any old icon is destroyed when SB.SetIcon() replaces it with a new one. This can be avoided via: MyGui.Opt("+LastFound") SendMessage(0x040F, part_number - 1, my_hIcon, "msctls_statusbar321") ; 0x040F is SB_SETICON. SetProgress Creates and controls a progress bar inside the status bar. This function is available at www.autohotkey.com/forum/topic37754.html StatusBar Usage Reacting to mouse clicks: Whenever the user clicks on the bar, the Click, DoubleClick or

ContextMenu event is raised, and the Info or Item parameter contains the part number. However, the part number might be a very large integer if the user clicks near the sizing grip at the right side of the bar. Font and color: Although the font size, face, and style can be set via MyGui.SetFont (just like normal controls), the text color cannot be changed. The status bar's background color may be changed by specifying in Options the word Background followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: BackgroundSilver, BackgroundFFDD99, BackgroundDefault. Note that the control must have Classic Theme appearance. Thus, the -Theme option must be specified along with the Background option, e.g. -Theme BackgroundSilver. Hiding the StatusBar: Upon creation, the bar can be hidden via MyStatusBar := MyGui.Add("StatusBar", "Hidden"). To hide it sometime after creation, use MyStatusBar.Visible := false. To show it, use MyStatusBar.Visible := true. Note: Hiding the bar does not reduce the height of the window. If that is desired, one easy way is MyGui.Show("AutoSize"). Styles (rarely used): See the StatusBar styles table. Known limitations: 1) Any control that overlaps the status bar might sometimes get drawn on top of it. One way to avoid this is to dynamically shrink such controls via Size event. 2) There is a limit of one status bar per window. Example: Example #1 at the bottom of the TreeView page demonstrates a multipart status bar. ActiveX ActiveX components such as the MSIE browser control can be embedded into a GUI window as follows. For details about the ActiveX component and its method used below, see WebBrowser object (Microsoft Docs) and Navigate method (Microsoft Docs). MyGui := Gui() WB := MyGui.Add("ActiveX", "w980 h640", "Shell.Explorer").Value ; The last parameter is the name of the ActiveX component. WB.Navigate("https://www.autohotkey.com/docs/") ; This is specific to the web browser control. MyGui.Show() When the control is created, the ActiveX object can be retrieved via GuiCtrl.Value. To handle events exposed by the object, use ComObjConnect as follows. For details about the event used below, see NavigateComplete2 event (Microsoft Docs). MyGui := Gui() URL := MyGui.Add("Edit", "w930 r1", "https://www.autohotkey.com/docs/") MyGui.Add ("Button", "x+6 yp w44 Default", "Go").OnEvent("Click", ButtonGo) WB := MyGui.Add("ActiveX", "xm w980 h640", "Shell.Explorer").Value ComObjConnect (WB, WB_events) ; Connect WB's events to the WB_events class object. MyGui.Show() ; Continue on to load the initial page: ButtonGo() ButtonGo(*) { WB.Navigate(URL.Value) } class WB_events { static NavigateComplete2(wb, &NewURL, *) { URL.Value := NewURL ; Update the URL edit control. } } ComObjType can be used to determine the type of the retrieved object. Custom Other controls which are not directly supported by AutoHotkey can be also embedded into a GUI window. In order to do so, include in Options the word Class followed by the Win32 class name of the desired control. Examples: MyGui.Add ("Custom", "ClassComboBoxEx32") ; Adds a ComboBoxEx control. MyGui.Add("Custom", "ClassScintilla") ; Adds a Scintilla control. Note that the SciLexer.dll library must be loaded before the control can be added. AutoHotkey uses the standard Windows control text routines when text is to be retrieved/replaced in the control via MyGui.Add or GuiCtrl.Value. Events: Since the meaning of each notification code depends on the control which sent it, OnEvent is not supported for Custom controls. However, if the control sends notifications in the form of a WM_NOTIFY or WM_COMMAND message, the script can use OnNotify or OnCommand to detect them. Here is an example that shows how to add and use an IP address control: MyGui := Gui() IP := MyGui.Add("Custom", "ClassSysIPAddress32 r1 w150") IP.OnCommand(0x300, IP_EditChange) ; 0x300 = EN_CHANGE IP.OnNotify(-860, IP_FieldChange) ; -860 = IPN_FIELDCHANGED IPText := MyGui.Add("Text", "wp") IPField := MyGui.Add("Text", "wp y+m") MyGui.Add("Button", "Default", "OK").OnEvent ("Click", OK_Click) MyGui.Show() IPCtrlSetAddress(IP, SysGetIPAddresses()[1]) OK_Click(*) { MyGui.Hide() MsgBox("You chose " IPCtrlGetAddress(IP)) ExitApp() } IP_EditChange(*) { IPText.Text := "New text: " IP.Text } IP_FieldChange(thisCtrl, NMIPAddress) { ; Extract info from the NMIPAddress structure. iField := NumGet(NMIPAddress, 3*A_PtrSize + 0, "int") iValue := NumGet(NMIPAddress, 3*A_PtrSize + 4, "int") if (iValue >= 0) IPField.Text := "Field #" iField " modified: " iValue else IPField.Text := "Field #" iField " left empty" } IPCtrlSetAddress(GuiCtrl, IPAddress) { static WM_USER := 0x0400 static IPM_SETADDRESS := WM_USER + 101 ; Pack the IP address into a 32-bit word for use with SendMessage. IPAddrWord := 0 Loop Parse IPAddress, "." IPAddrWord := (IPAddrWord * 256) + A_LoopField SendMessage(IPM_SETADDRESS, 0, IPAddrWord, GuiCtrl) } IPCtrlGetAddress(GuiCtrl) { static WM_USER := 0x0400 static IPM_GETADDRESS := WM_USER + 102 AddrWord := Buffer(4) SendMessage(IPM_GETADDRESS, 0, AddrWord, GuiCtrl) IPPart := [] Loop 4 IPPart.Push(NumGet(AddrWord, 4 - A_Index, "UChar")) return IPPart[1] "." IPPart[2] "." IPPart[3] "." IPPart[4] } Related Pages ListView, TreeView, Gui(), Gui object, GuiControl object, Menu object GuiCtrlFromHwnd - Syntax & Usage | AutoHotkey v2 GuiCtrlFromHwnd Retrieves the GuiControl object of a GUI control associated with the specified HWND. GuiControlObj := GuiCtrlFromHwnd(Hwnd) Parameters Hwnd Type: Integer The window handle (HWND) of a GUI control, or a child window of such a control (e.g. the Edit control of a ComboBox). The control must have been created by the current script, by calling Add or AddControlType. Return Value Type: GuiControl or String (empty) This function returns the GuiControl object associated with the specified HWND, or an empty string if there isn't one or the HWND is invalid. Remarks For example, a HWND of a GUI control can be retrieved via GuiCtrl.Hwnd, MouseGetPos or OnMessage. Related Gui(), Gui object, GuiControl object, GuiFromHwnd, Control Types, ListView, TreeView, Menu object, Control functions, MsgBox, FileSelect, DirSelect Examples See the ToolTip example on the Gui object page. GuiFromHwnd - Syntax & Usage | AutoHotkey v2 GuiFromHwnd Retrieves the Gui object of a GUI window associated with the specified HWND. GuiObj := GuiFromHwnd(Hwnd , RecurseParent) Parameters Hwnd Type: Integer The window handle (HWND) of a GUI window previously created by the script, or if RecurseParent is true, any child window of a GUI window created by the script. RecurseParent Type: Boolean If this parameter is true and Hwnd identifies a child window which is not a GUI, the function searches for and retrieves its closest parent window which is a GUI. Otherwise, the function returns an empty string if Hwnd does not directly identify a GUI window. Return Value Type: Gui or String (empty) This function returns the Gui object associated with the specified HWND, or an empty string if there isn't one or the HWND is invalid. Remarks For example, the HWND of a GUI window may be passed to an OnMessage function, or can be retrieved via Gui.Hwnd, WinExist or some other method. Related Gui(), Gui object, GuiControl object, GuiCtrlFromHwnd, Control Types, ListView, TreeView, Menu object, Control functions, MsgBox, FileSelect, DirSelect Examples Retrieves the Gui object by using the HWND of the GUI window just created and reports its title. MyGui := Gui(, "Title of Window") MyGui.Add ("Text",, "Some text to display.") MyGui.Show() MsgBox(GuiFromHwnd(MyGui.Hwnd).Title) OnCommand (GUI) - Syntax & Usage | AutoHotkey v2 OnCommand Registers a function or method to be called when a control notification is received via the WM_COMMAND message. GuiCtrl.OnCommand (NotifyCode, Callback , AddRemove) Parameters NotifyCode Type: Integer The control-defined notification code to monitor. Callback Type: String or Function Object The function, method or object to call when the event is raised. If the GUI has an event sink (that is, if Gui()'s EventObj parameter was specified), this parameter may be the name of a method belonging to the event sink. Otherwise, this parameter must be a function object. AddRemove Type: Integer If omitted, it defaults to 1 (call the callback after any previously registered callbacks). Otherwise, specify one of the following numbers: 1 = Call the callback after any previously registered callbacks. -1 = Call the callback before any previously registered callbacks. 0 = Do not call the callback. WM_COMMAND Certain types of controls send a WM_COMMAND message whenever an interesting event occurs. These are usually standard Windows controls which have been around a long time, as newer controls use the WM_NOTIFY message (see OnNotify). Commonly used notification codes are translated to events, which the script can monitor with OnEvent. The message's parameters contain the control ID, HWND and notification code, which AutoHotkey uses to dispatch the notification to the appropriate callback. There are no additional parameters. To determine which notifications are available (if any), refer to the control's documentation. Control Library (Microsoft Docs) contains links to each of the the Windows common controls (however, only a few of these use WM_COMMAND). The notification codes (numbers) can be found in the Windows SDK, or by searching the Internet. Callback Parameters The notes for OnEvent regarding this and bound functions also apply to OnCommand. The callback receives one parameter: Callback(GuiControl) Callback Return Value If multiple callbacks have been registered for an event, a callback may return a non-empty value to prevent any remaining callbacks from being called. The return value is ignored by the control. Related These notes for OnEvent also apply to OnCommand: Threads, Destroying the GUI. OnNotify can be used for notifications which are sent as a WM_NOTIFY message. OnEvent (GUI) - Syntax & Usage | AutoHotkey v2 OnEvent Registers a function or method to be called when the given event is raised by a GUI window or control. Gui.OnEvent(EventName, Callback , AddRemove) GuiCtrl.OnEvent(EventName, Callback , AddRemove) Parameters EventName Type: String The name of the event. See Events further below. Callback Type: String or Function Object The function, method or object to call when the event is raised. If the GUI has an event sink (that is, if Gui()'s EventObj parameter was specified), this parameter may be the name of a method belonging to the event sink. Otherwise, this parameter must be a function object. AddRemove Type: Integer If omitted, it defaults to 1 (call the callback after any previously registered callbacks). Otherwise, specify one of the following numbers: 1 = Call the callback after any previously registered callbacks. -1 = Call the callback before any previously registered callbacks. 0 = Do not call the callback. Callback Parameters If the callback is a method registered by name, its hidden this parameter seamlessly receives the event sink object (that is, the object to which the method belongs). This parameter is not shown in the parameter lists in this documentation. Since Callback can be an object, it can be a BoundFunc object which inserts additional parameters at the beginning of the parameter list and then calls another function. This is a general technique not specific to OnEvent, so is generally ignored by the rest of this documentation. The callback's first explicit parameter is the Gui or GuiControl object which raised the event. The only exception is that this parameter is omitted when a Gui handles its own events, since this already contains a reference to the Gui. Many events pass additional parameters about the event, as described for each event. As with all methods or functions called dynamically, the callback is not required to declare parameters which the callback itself does not need, but in this case an asterisk must be specified as the final parameter. If an event has more parameters than are declared by the callback, they will simply be ignored (unless the callback is variadic). The callback can declare more parameters than the event provides if (and only if) the additional parameters are declared optional. However, the use of optional parameters is not recommended as future versions of the program may extend an event with additional parameters, in which case the optional parameters would stop receiving their default values. Callback Return Value If multiple callbacks have been registered for an event, a callback may return a non-empty value to prevent any remaining callbacks from being called. The return value may have additional meaning for specific events. For example, a Close callback may return a non-zero number (such as true) to prevent the GUI window from closing. Callback Name By convention, the syntax of each event below is shown with a function name of the form ObjectType_EventName, for clarity. Scripts are not required to follow this

convention, and can use any valid function name. Threads Each event callback is called in a new thread, and therefore starts off fresh with the default values for settings such as SendMode. These defaults can be changed during script startup. Whenever a GUI thread is launched, that thread's last found window starts off as the GUI window itself. This allows functions for windows and controls -- such as WinGetStyle, WinSetTransparent, and ControlGetFocus -- to omit WinTitle and WinText when operating upon the GUI window itself (even if it is hidden). Except where noted, each event is limited to one thread at a time, per object. If an event is raised before a previous thread started by that event finishes, it is usually discarded. To prevent this, use Critical as the callback's first line (however, this will also buffer/defer other threads such as the press of a hotkey). Destroying the GUI When a GUI is destroyed, all event callbacks are released. Therefore, if the GUI is destroyed while an event is being dispatched, subsequent event callbacks are not called. For clarity, callbacks should return a non-empty value after destroying the GUI. Events The following events are supported by Gui objects: EventRaised when... CloseThe window is closed. ContextMenuThe user right-clicks within the window or presses Menu or Shift+F10. DropFilesFiles/folders are dragged and dropped onto the window. EscapeThe user presses Esc while the GUI window is active. SizeThe window is resized, minimized, maximized or restored. The following events are supported by GuiControl objects, depending on the control type: EventRaised when... ChangeThe control's value changes. ClickThe control is clicked. DoubleClickThe control is double-clicked. ColClickOne of the ListView's column headers is clicked. ContextMenuThe user right-clicks the control or presses Menu or Shift+F10 while the control has the keyboard focus. FocusThe control gains the keyboard focus. LoseFocusThe control loses the keyboard focus. ItemCheckA ListView or TreeView item is checked or unchecked. ItemEditA ListView or TreeView item's label is edited by the user. ItemExpandA TreeView item is expanded or collapsed. ItemFocusThe focused item changes in a ListView. ItemSelectA ListView or TreeView item is selected, or a ListView item is deselected. Window Events Close Launched when the user or another program attempts to close the window, such as by pressing the X button in its title bar, selecting "Close" from its system menu, or calling WinClose. Gui_Close(GuiObj) By default, the window is automatically hidden after the callback returns, or if no callbacks were registered. A callback can prevent this by returning 1 (or true), which will also prevent any remaining callbacks from being called. The callback can hide the window immediately by calling Gui.Hide, or destroy the window by calling Gui.Destroy. For example, this GUI shows a confirmation prompt before closing: myGui := Gui() myGui.AddText("", "Press Alt+F4 or the X button in the title bar.") myGui.OnEvent("Close", myGui_Close) myGui_Close(thisGui) { ; Declaring this parameter is optional. if MsgBox("Are you sure you want to close the GUI?",, "y/n") = "No" return true ; true = 1 } myGui.Show ContextMenu Launched whenever the user right-clicks anywhere in the window except the title bar and menu bar. It is also launched in response to pressing Menu or Shift+F10. Gui_ContextMenu(GuiObj, GuiCtrlObj, Item, IsRightClick, X, Y) GuiCtrlObj Type: GuiControl or String (empty) The object of the control that received the event (blank if none). Item Type: Integer When a ListBox, ListView, or TreeView is the target of the context menu (as determined by GuiCtrlObj), Item specifies which of the control's items is the target. ListBox: The number of the currently focused row. Note that a standard ListBox does not focus an item when it is right-clicked, so this might not be the clicked item. ListView and TreeView: For right-clicks, Item contains the clicked item's ID or row number (or 0 if the user clicked somewhere other than an item). For the AppsKey and Shift-F10, Item contains the selected item's ID or row number. IsRightClick Type: Integer (boolean) True if the user clicked the right mouse button. False if the user pressed Menu or Shift+F10. X, Y Type: Integer The X and Y coordinates of where the script should display the menu (e.g. MyContextMenu.Show X, Y). Coordinates are relative to the upper-left corner of the window's client area. Unlike most other GUI events, the ContextMenu event can have more than one concurrent thread. Each control can have its own ContextMenu event callback which is called before any callback registered for the Gui object. Control-specific callbacks omit the GuiObj parameter, but all other parameters are the same. Note: Since Edit and MonthCal controls have their own context menu, a right-click in one of them will not launch the ContextMenu event. DropFiles Launched whenever files/folders are dropped onto the window as part of a drag-and-drop operation (but if this callback is already running, drop events are ignored). Gui_DropFiles(GuiObj, GuiCtrlObj, FileArray, X, Y) GuiCtrlObj Type: GuiControl or String (empty) The object of the control upon which the files were dropped (blank if none). FileArray Type: Array of Strings An array of filenames, where FileArray[1] is the first file and FileArray.Length returns the number of files. A for-loop can be used to iterate through the files: Gui_DropFiles(GuiObj, GuiCtrlObj, FileArray, X, Y) { for i, DroppedFile in FileArray MsgBox "File " i " is:`n" DroppedFile } X, Y Type: Integer The X and Y coordinates of where the files were dropped, relative to the upper-left corner of the window's client area. Escape Launched when the user presses Esc while the GUI window is active. Gui_Escape(GuiObj) By default, pressing Esc has no effect. Known limitation: If the first control in the window is disabled (possibly depending on control type), the Escape event will not be launched. There may be other circumstances that produce this effect. Size Launched when the window is resized, minimized, maximized, or restored. Gui_Size (GuiObj, MinMax, Width, Height) MinMax Type: Integer One of the following values: 0 = The window is neither minimized nor maximized. 1 = The window is maximized. -1 = The window is minimized. Note that a maximized window can be resized without restoring/un-maximizing it, so a value of 1 does not necessarily mean that this event was raised in response to the user maximizing the window. Width, Height Type: Integer The new width and height of the window's client area, which is the area excluding title bar, menu bar, and borders. A script may use the Size event to reposition and resize controls in response to the user's resizing of the window. When the window is resized (even by the script), the Size event might not be raised immediately. As with other window events, if the current thread is uninterruptible, the Size event won't be raised until the thread becomes interruptible. If the script has just resized the window, follow this example to ensure the Size event is raised immediately: Critical "Off" ; Even if Critical "On" was never used. Sleep -1 Gui.Show automatically does a Sleep -1, so it is generally not necessary to call Sleep in that case. Control Events Change Raised when the control's value changes. Ctrl_Change(GuiCtrlObj, Info) Info Type: Integer For Slider controls, Info is a numeric value indicating how the slider moved. For details, see Detecting Changes (Slider). For all other controls, Info currently has no meaning. To retrieve the control's new value, use GuiCtrlObj.Value. Applies to: DDL, ComboBox, ListBox, Edit, DateTime, MonthCal, Hotkey, UpDown, Slider, Tab. Click Raised when the control is clicked. Ctrl_Click(GuiCtrlObj, Info) Link_Click(GuiCtrlObj, Info, Href) Info Type: Integer ListView: The row number of the clicked item, or 0 if the mouse was not over an item. TreeView: The ID of the clicked item, or 0 if the mouse was not over an item. Link: The link's ID attribute (a string) if it has one, otherwise the link's index (an integer). StatusBar: The part number of the clicked section (however, the part number might be a very large integer if the user clicks near the sizing grip at the right side of the bar). For all other controls, Info currently has no meaning. Href Type: String Link: The link's HREF attribute. Note that if a Click event callback is registered, the HREF attribute is not automatically executed. Applies to: Text, Pic, Button, CheckBox, Radio, ListView, TreeView, Link, StatusBar. DoubleClick Raised when the control is double-clicked. Ctrl_DoubleClick(GuiCtrlObj, Info) Info Type: Integer ListView, TreeView and StatusBar: Same as for the Click event. ListBox: The position of the focused item. Double-clicking empty space below the last item usually focuses the last item and leaves the selection as it was. Applies to: Text, Pic, Button, CheckBox, Radio, ComboBox, ListBox, ListView, TreeView, StatusBar. ColClick Raised when one of the ListView's column headers is clicked. Ctrl_ColClick(GuiCtrlObj, Info) Info Type: Integer The one-based column number that was clicked. This is the original number assigned when the column was created; that is, it does not reflect any dragging and dropping of columns done by the user. Applies to: ListView. ContextMenu Raised when the user right-clicks the control or presses Menu or Shift+F10 while the control has the keyboard focus. Ctrl_ContextMenu(GuiCtrlObj, Item, IsRightClick, X, Y) For details, see ContextMenu. Applies to: All controls except Edit and MonthCal (and the Edit control within a ComboBox), which have their own standard context menu. Focus / LoseFocus Raised when the control gains or loses the keyboard focus. Ctrl_Focus(GuiCtrlObj, Info) Ctrl_LoseFocus (GuiCtrlObj, Info) Info Reserved. Applies to: Button, CheckBox, Radio, DDL, ComboBox, ListBox, ListView, TreeView, Edit, DateTime. Not supported: Hotkey, Slider, Tab and Link. Note that Text, Pic, MonthCal, UpDown and StatusBar controls do not accept the keyboard focus. ItemCheck Raised when a ListView or TreeView item is checked or unchecked. Ctrl_ItemCheck(GuiCtrlObj, Item, Checked) Applies to: ListView, TreeView. ItemEdit Raised when a ListView or TreeView item's label is edited by the user. Ctrl_ItemEdit(GuiCtrlObj, Item) An item's label can only be edited if -ReadOnly has been used in the control's options. Applies to: ListView, TreeView. ItemExpand Raised when a TreeView item is expanded or collapsed. Ctrl_ItemExpand(GuiCtrlObj, Item, Expanded) Applies to: TreeView. ItemFocus Raised when the focused item changes in a ListView. Ctrl_ItemFocus(GuiCtrlObj, Item) Applies to: ListView. ItemSelect Raised when a ListView or TreeView item is selected, or a ListView item is deselected. ListView_ItemSelect(GuiCtrlObj, Item, Selected) TreeView_ItemSelect(GuiCtrlObj, Item) Applies to: ListView, TreeView. ListView: This event is raised once for each item being deselected or selected, so can be raised multiple times in response to a single action by the user. Other Events Other types of GUI events can be detected and acted upon via OnNotify, OnCommand or OnMessage. For example, a script can display context-sensitive help via ToolTip whenever the user moves the mouse over particular controls in the window. This is demonstrated in the GUI ToolTip example. OnNotify (GUI) - Syntax & Usage | AutoHotkey v2 OnNotify Registers a function or method to be called when a control notification is received via the WM_NOTIFY message. GuiCtrl.OnNotify(NotifyCode, Callback , AddRemove) Parameters NotifyCode Type: Integer The control-defined notification code to monitor. Callback Type: String or Function Object The function, method or object to call when the event is raised. If the GUI has an event sink (that is, if Gui()'s EventObj parameter was specified), this parameter may be the name of a method belonging to the event sink. Otherwise, this parameter must be a function object. AddRemove Type: Integer If omitted, it defaults to 1 (call the callback after any previously registered callbacks). Otherwise, specify one of the following numbers: 1 = Call the callback after any previously registered callbacks. -1 = Call the callback before any previously registered callbacks. 0 = Do not call the callback. WM_NOTIFY Certain types of controls send a WM_NOTIFY message whenever an interesting event occurs or the control requires information from the program. The lParam parameter of this message contains a pointer to a structure containing information about the notification. The type of structure depends on the notification code and the type of control which raised the notification, but is always based on NMHDR. To determine which notifications are available (if any), what type of structure they provide and how they interpret the return value, refer to the control's documentation. Control Library (Microsoft Docs) contains links to each of the the Windows common controls. The notification codes (numbers) can be found in the Windows SDK, or by searching the Internet. AutoHotkey uses the idFrom and hwndFrom fields to identify which control sent the notification, in order to dispatch it to the appropriate object. The code field contains the notification code. Since these must match up to the GuiControl and NotifyCode used to register the callback, they are rarely useful to the script. Callback Parameters The notes

for OnEvent regarding this and bound functions also apply to OnNotify. The callback receives two parameters: Callback(GuiControl, lParam) As noted above, lParam is the address of a notification structure derived from NMHDR. Its exact type depends on the type of control and notification code. If the structure contains additional information about the notification, the callback can retrieve it with NumGet and/or StrGet. Callback Return Value If multiple callbacks have been registered for an event, a callback may return a non-empty value to prevent any remaining callbacks from being called. The return value may have additional meaning, depending on the notification. For example, the ListView notification LVN_BEGINLABELEDIT (-175 or -105) prevents the user from editing the label if the callback returns TRUE (1). Related These notes for OnEvent also apply to OnNotify: Threads, Destroying the GUI. OnCommand can be used for notifications which are sent as a WM_COMMAND message. HasBase - Syntax & Usage | AutoHotkey v2 HasBase Returns a non-zero number if the specified value is derived from the specified base object. HasBase := HasBase(Value, BaseObj) Parameters Value Any value, of any type. BaseObj Type: Object The potential base object to test. Return Value Type: Integer (boolean) This function returns true if BaseObj is in Value's chain of base objects, otherwise false. Remarks The following code is roughly equivalent to this function: MyHasBase(Value, BaseObj) { b := Value while b := ObjGetBase(b) if b = BaseObj return true return false } For example, HasBase(Obj, Array.Prototype) is true if Obj is an instance of Array or any derived class. This the same check performed by Obj is Array; however, instances can be based on other instances, whereas is requires a Class. HasBase accepts both objects and primitive values. For example, HasBase(1, 0.base) returns true. Related Objects, Obj.Base, ObjGetBase, HasMethod, HasProp Examples Illustrates the use of this function. thebase := {key: "value"} derived := {base: thebase} MsgBox HasBase(thebase, derived) ; 0 MsgBox HasBase(derived, thebase) ; 1 HasMethod - Syntax & Usage | AutoHotkey v2 HasMethod Returns a non-zero number if the specified value has a method by the specified name. HasMethod := HasMethod(Value , Name, ParamCount) Parameters Value Type: Any Any value, of any type except ComObject. Name Type: String The method name to check for. Omit this parameter to check whether Value itself is callable. ParamCount Type: Integer The number of parameters that would be passed to the method or function. If specified, the method's MinParams, MaxParams and IsVariadic properties may be queried to verify that it can accept this number of parameters. If those properties are not present, the parameter count is not verified. This count should not include the implicit this parameter. If omitted (or if the parameter count was not verified), a basic check is performed for a Call method to verify that the object is most likely callable. Return Value Type: Integer (boolean) This function returns true if a method was found and passed validation (if performed), otherwise false. Remarks HasMethod has the same limitations as GetMethod. This function can be used to estimate whether a value supports a specific action. For example, values without a Call method cannot be called or passed to SetTimer, while values without either an __Enum method or a Call method cannot be passed to For. However, the existence of a method does not guarantee that it can be called, since there are requirements that must be met, such as parameter count. When ParamCount is specified, the validation this function performs is equivalent to the validation performed by built-in functions such as SetTimer. A return value of false does not necessarily indicate that the method cannot be called, as the value may have a __Call meta-function. However, __Call is not triggered in certain contexts, such as when __Enum is being called by For. If __Call is present, there is no way to detect which methods it may support. This function supports primitive values. Related Objects, HasBase, HasProp, GetMethod Examples Illustrates the use of this function. MsgBox HasMethod(0, "HasMethod") ; 1 MsgBox HasMethod(0, "Call") ; 0 HasProp - Syntax & Usage | AutoHotkey v2 HasProp Returns a non-zero number if the specified value has a property by the specified name. HasProp := HasProp (Value, Name) Parameters Value Type: Any Any value, of any type except ComObject. Name Type: String The property name to check for. Return Value Type: Integer (boolean) This function returns true if the value has a property by this name, otherwise false. Remarks This function does not test for the presence of a __Get or __Set meta-function. If present, there is no way to detect the exact set of properties that it may implement. This function supports primitive values. Related Objects, HasBase, HasMethod Examples Illustrates the use of this function. MsgBox HasProp({}, "x") ; 0 MsgBox HasProp({x:1}, "x") ; 1 MsgBox HasProp(0, "Base") ; 1 HotIf - Syntax & Usage | AutoHotkey v2 HotIf / HotIfWin Specifies the criteria for subsequently created or modified hotkey variants. Contents: HotIf HotIfWin... General Remarks Examples HotIf HotIf "Expression" HotIf Function Parameters (parameter omitted) Sets blank criteria (turns off context-sensitivity). "Expression" Type: String Sets the criteria to an existing #HotIf expression. Expression is usually written as a quoted string, but can also be a variable or expression which returns text matching a #HotIf expression. Although this function is unable to create new expressions, it can create new hotkeys using an existing expression. See #HotIf example 5. Note: The HotIf function uses the string that you pass to it, not the original source code. Escape sequences are resolved when the script loads, so only the resulting characters are considered; for example, HotIf 'x = "`t"' and HotIf 'x = "' A_Tab '"' both correspond to #HotIf x = "`t". Function Type: Function Object Sets the criteria to a given function object. Subsequently-created hotkeys will only execute if calling the given function object yields a non-zero number. This is like HotIf "Expression", except that each hotkey can have many variants (one per object). Function must be an object with a call method taking one parameter, the name of the hotkey. Once passed to the HotIf function, the object will never be deleted (but memory will be reclaimed by the OS when the process exits). The "three-key combination" Hotkey example uses this mode of HotIf. HotIfWin... HotIfWinActive WinTitle, WinText HotIfWinExist WinTitle, WinText HotIfWinNotActive WinTitle, WinText HotIfWinNotExist WinTitle, WinText Parameters (all parameters omitted) Sets blank criteria (turns off context-sensitivity). WinTitle, WinText Type: String Specifies the window title and other criteria that should be used to identify a window. Depending on which function is called, affected hotkeys and hotstrings are active only while a matching window is active, exists, is not active, or does not exist. Since the parameters are evaluated before the function is called, any variable reference becomes permanent at that moment. In other words, subsequent changes to the contents of the variable are not seen by existing hotkeys. WinTitle and WinText have the same meaning as for WinActive or WinExist, but only strings can be used, and they are evaluated according to the default settings for SetTitleMatchMode and DetectHiddenWindows as set by the auto-execute thread. See WinTitle for details. Error Handling An exception is thrown if HotIf's parameter is invalid, such as if it does not match an existing expression and is not a valid callback function. General Remarks HotIf allows context-sensitive hotkeys to be created and modified while the script is running (by contrast, the #HotIf directive is positional and takes effect before the script begins executing). For example: HotIfWinActive "ahk_class Notepad" Hotkey "^!e", MyFuncForNotepad ; Creates a hotkey that works only in Notepad. Using HotIf puts context sensitivity into effect for all subsequently created hotkeys in the current thread, and affects which hotkey variants the Hotkey function modifies. Only the most recent call to any of the HotIf functions in the current thread will be in effect. To turn off context sensitivity (such as to make subsequently-created hotkeys work in all windows), call any HotIf function but omit the parameters. For example: HotIf or HotIfWinActive. Before HotIf is used in a hotkey or hotstring thread, the Hotkey and Hotstring functions default to the same context as the hotkey or hotstring that launched the thread. In other words, Hotkey A_ThisHotkey, "Off" turns off the current hotkey even if it is context-sensitive. All other threads default to creating or modifying global hotkeys, unless that default is overridden by using HotIf during script startup. When a mouse or keyboard hotkey is disabled via a HotIf function or directive, it performs its native function; that is, it passes through to the active window as though there is no such hotkey. However, joystick hotkeys always pass through, whether they are disabled or not. Related Hotkey (function), Hotkeys (general), #HotIf, Threads Examples See Hotkey for examples. Hotkey - Syntax & Usage | AutoHotkey v2 Hotkey Creates, modifies, enables, or disables a hotkey while the script is running. Hotkey KeyName , Action, Options Parameters KeyName Type: String Name of the hotkey's activation key, including any modifier symbols. For example, specify #c for the Win+C hotkey. If KeyName already exists as a hotkey, that hotkey will be updated with the values of the function's other parameters. KeyName can also be the name of an existing hotkey (created with the double-colon syntax), which will cause that hotkey to be updated with the values of the function's other parameters. When specifying an existing hotkey, KeyName is not case sensitive. However, the names of keys must be spelled the same as in the existing hotkey (e.g. Esc is not the same as Escape for this purpose). Also, the order of modifier symbols such as ^!+# does not matter. GetKeyName can be used to retrieve the standard spelling of a key name. 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 as reflected by A_ThisHotkey and Action's parameter. This name is shared by all variants of the hotkey, and does not change even if the Hotkey function later accesses the hotkey with a different symbol ordering. If the hotkey variant already exists, its behavior is updated according to whether KeyName includes or excludes the tilde (~) prefix. The use hook ($) prefix can be added to existing hotkeys. This prefix affects all variants of the hotkey and cannot be removed. Action Type: String or Function Object An action name or a function to execute (as a new thread) when the hotkey is pressed. This can be a function object, a hotkey name without trailing colons, or one of the special values listed below. If Action is a function, it is called with one parameter, the name of the hotkey. Hotkeys defined with the double-colon syntax automatically use the parameter name ThisHotkey. Hotkeys can also be assigned a function name without the Hotkey function. If Action is a hotkey name, its original function is used; specifically, the original function of the hotkey variant corresponding to the current HotIf criteria. This is usually used to restore a hotkey's original function after having changed it, but can be used to assign the function of a different hotkey, provided that both hotkeys use the same HotIf criteria. This parameter can be left blank if KeyName already exists as a hotkey, in which case its action will not be changed. This is useful to change only the hotkey's Options. Note: If the function is specified but the hotkey is disabled from a previous use of this function, the hotkey will remain disabled. To prevent this, include the word ON in Options. This parameter can also be one of the following special values: On: The hotkey becomes enabled. No action is taken if the hotkey is already On. Off: The hotkey becomes disabled. No action is taken if the hotkey is already Off. Toggle: The hotkey is set to the opposite state (enabled or disabled). AltTab (and others): These are special Alt-Tab hotkey actions that are described here. Options Type: String A string of zero or more of the following letters with optional spaces in between. For example: On B0. On: Enables the hotkey if it is currently disabled. Off: Disables the hotkey if it is currently enabled. This is typically used to create a hotkey in an initially-disabled state. B or B0: Specify the letter B to buffer the hotkey as described in #MaxThreadsBuffer. Specify B0 (B with the number 0) to disable this type of buffering. Pn: Specify the letter P followed by the hotkey's thread priority. If the P option is omitted when creating a hotkey, 0 will be used. S or S0: Specify the letter S to make the hotkey exempt from Suspend, which allows the hotkey to be used to turn Suspend off. Specify S0 (S with the number 0) to remove the exemption, allowing the hotkey to be suspended. Tn: Specify the letter T followed by a the number of threads to allow for this hotkey as described in #MaxThreadsPerHotkey. For example: T5. In (InputLevel): Specify the letter I (or i) followed by the hotkey's input level. For example: I1. If

any of the option letters are omitted and the hotkey already exists, those options will not be changed. But if the hotkey does not yet exist -- that is, it is about to be created by this function -- the options will default to those most recently in effect. For example, the instance of #MaxThreadsBuffer that occurs closest to the bottom of the script will be used. If #MaxThreadsBuffer does not appear in the script, its default setting (OFF in this case) will be used. Error Handling An exception is thrown if a parameter is invalid or memory allocation fails. One of the following exceptions may be thrown if the hotkey is invalid or could not be created: Error Class Message Description ValueError Invalid key name. The KeyName parameter specifies one or more keys that are either not recognized or not supported by the current keyboard layout/language. Exception.Extra contains the key name; e.g. "Entre" from !Entre. Unsupported prefix key. For example, using the mouse wheel as a prefix in a hotkey such as WheelDown & Enter is not supported. Exception.Extra contains the prefix key. This AltTab hotkey must have exactly one modifier/prefix The KeyName parameter is not suitable for use with the AltTab or ShiftAltTab actions. A combination of (at most) two keys is required. For example: RControl & RShift::AltTab. Exception.Extra contains KeyName. This AltTab hotkey must specify which key (L or R). TargetError Nonexistent hotkey. The function attempted to modify a nonexistent hotkey. Exception.Extra contains KeyName. Nonexistent hotkey variant (IfWin). The function attempted to modify a nonexistent variant of an existing hotkey. To solve this, use HotIf to set the criteria to match those of the hotkey to be modified. Exception.Extra contains KeyName. Error Max hotkeys. Creating this hotkey would exceed the limit of 32762 hotkeys per script (however, each hotkey can have an unlimited number of variants, and there is no limit to the number of hotstrings). Tip: Try-Catch can be used to test for the existence of a hotkey variant. For example: try Hotkey "^!p" catch TargetError MsgBox "The hotkey does not exist or it has no variant for the current If criteria." Remarks The current HotIf setting determines the variant of a hotkey upon which the Hotkey function will operate. If the goal is to disable selected hotkeys or hotstrings automatically based on the type of window that is active, Hotkey "^!c", "Off" is usually less convenient than using #HotIf with WinActive/WinExist (or their dynamic counterparts HotIfWinActive/Exist). Creating hotkeys via the double-colon syntax performs better than using the Hotkey function because the hotkeys can all be enabled as a batch when the script starts (rather than one by one). Therefore, it is best to use this function to create only those hotkeys whose key names are not known until after the script has started running. One such case is when a script's hotkeys for various actions are configurable via an INI file. If the script is suspended, newly added/enabled hotkeys will also be suspended until the suspension is turned off (unless they are exempt as described in the Suspend section). The keyboard and/or mouse hooks will be installed or removed if justified by the changes made by this function. Although the Hotkey function cannot directly enable or disable hotkeys in scripts other than its own, in most cases it can override them by creating or enabling the same hotkeys. Whether this works depends on a combination of factors: 1) Whether the hotkey to be overridden is a hook hotkey in the other script (non-hook hotkeys can always be overridden); 2) The fact that the most recently started script's hotkeys generally take precedence over those in other scripts (therefore, if the script intending to override was started most recently, its override should always succeed); 3) Whether the enabling or creating of this hotkey will newly activate the keyboard or mouse hook (if so, the override will always succeed). Once a script has at least one hotkey, it becomes persistent, meaning that ExitApp rather than Exit should be used to terminate it. Variant (Duplicate) Hotkeys A particular hotkey can be created more than once if each definition has different If criteria. These are known as hotkey variants. For example: HotIfWinActive "ahk_class Notepad" Hotkey "^!c", MyFuncForNotepad HotIfWinActive "ahk_class WordPadClass" Hotkey "^!c", MyFuncForWordPad HotIfWinActive Hotkey "^!c", MyFuncForAllOtherWindows If more than one variant of a hotkey is eligible to fire, only the one created earliest will fire. The exception to this is the global variant (the one with no If criteria): It always has the lowest precedence, and thus will fire only if no other variant is eligible. When creating duplicate hotkeys, the order of modifier symbols such as ^!+# does not matter. For example, "^!c" is the same as "!^c". However, keys must be spelled consistently. For example, Esc is not the same as Escape for this purpose (though the case does not matter). Finally, any hotkey with a wildcard prefix (*) is entirely separate from a non-wildcard one; for example, "*F1" and "F1" would each have their own set of variants. For more information about context-sensitive hotkeys, see #HotIf's General Remarks. Related Hotkey Symbols, HotIf, A_ThisHotkey, #MaxThreadsBuffer, #MaxThreadsPerHotkey, Suspend, Threads, Thread, Critical, Return, Menu object, SetTimer Examples Creates a Ctrl-Alt-Z hotkey. Hotkey "^!z", MyFunc MyFunc(ThisHotkey) { MsgBox "You pressed " ThisHotkey } Makes RCtrl & RShift operate like Alt-Tab. Hotkey "RCtrl & RShift", "AltTab" Disables the Shift-Win-C hotkey. Hotkey "$+#c", "Off" Changes a hotkey to allow 5 threads. Hotkey "^!a",, "T5" Creates Alt+W as a hotkey that works only in Notepad. HotIfWinActive "ahk_class Notepad" Hotkey "!w", ToggleWordWrap ; !w = Alt+W ToggleWordWrap (ThisHotkey) { MenuSelect "A",, "Format", "Word Wrap" } Creates a GUI that allows to register primitive three-key combination hotkeys. HkGui := Gui() HkGui.Add("Text", "xm", "Prefix key:") HkGui.Add("Edit", "yp x100 w100 vPrefix", "Space") HkGui.Add("Text", "xm", "Suffix hotkey:") HkGui.Add ("Edit", "yp x100 w100 vSuffix", "f & j") HkGui.Add("Button", "Default", "Register").OnEvent("Click", RegisterHotkey) HkGui.OnEvent("Close", (*) => ExitApp()) HkGui.OnEvent("Escape", (*) => ExitApp()) HkGui.Show() RegisterHotkey(*) { Saved := HkGui.Submit(false) HotIf (*) => GetKeyState(Saved.Prefix) Hotkey Saved.Suffix, (ThisHotkey) => MsgBox(ThisHotkey) } Hotstring - Syntax & Usage | AutoHotkey v2 Hotstring Creates, modifies, enables, or disables a hotstring while the script is running. Hotstring String , Replacement, OnOffToggle Hotstring NewOptions OldValue := Hotstring("EndChars" , NewValue) OldValue := Hotstring("MouseReset" , NewValue) Hotstring "Reset" Parameters String Type: String The hotstring's trigger string, preceded by the usual colons and option characters. For example, "::btw" or ":*:]d". String may be matched to an existing hotstring by considering case-sensitivity (C), word-sensitivity (?), activation criteria (as set by #HotIf or HotIf) and the trigger string. For example, "::btw" and "::BTW" match unless the case-sensitive mode was enabled as a default, while ":C:btw" and ":C:BTW" never match. The C and ? options may be included in String or set as defaults by the #Hotstring directive or a previous call to this function. If the hotstring already exists, any options specified in String are put into effect, while all other options are left as is. However, since hotstrings with C or ? are considered distinct from other hotstrings, it is not possible to add or remove these options. Instead, turn off the existing hotstring and create a new one. When a hotstring is first created -- either by the Hotstring function or the double-colon syntax in the script -- its trigger string and sequence of option characters becomes the permanent name of that hotstring as reflected by ThisHotkey. This name does not change even if the Hotstring function later accesses the hotstring with different option characters. Replacement Type: String or Function Object The replacement string, or a function or function object to call (as a new thread) when the hotstring triggers. If Replacement is a function, it is called with one parameter, the name of the hotstring. Hotstrings defined with the double-colon syntax automatically use the parameter name ThisHotkey. Hotstrings can also be assigned a function name without the Hotstring function. All strings are treated as replacement text. To make a hotstring call a function when triggered, pass the function by reference. Note that after reassigning the function of a hotstring, its original function can only be restored if it was given a name. This parameter can be omitted if the hotstring already exists, in which case its replacement will not be changed. This is useful to change only the hotstring's options, or to turn it on or off. Note: If this parameter is specified but the hotstring is disabled from a previous use of this function, the hotstring will remain disabled. To prevent this, include the word "On" in OnOffToggle. OnOffToggle Type: String or Integer One of the following values: On or 1 (true): Enables the hotstring. Off or 0 (false): Disables the hotstring. Toggle or -1: Sets the hotstring to the opposite state (enabled or disabled). Errors This function throws an exception if the parameters are invalid or a memory allocation fails. A TargetError is thrown if Replacement is omitted and String is valid but does not match an existing hotstring. This can be utilized to test for the existence of a hotstring. For example: try Hotstring "::btw" catch TargetError MsgBox "The hotstring does not exist or it has no variant for the current IfWin criteria." Remarks The current HotIf setting determines the variant of a hotstring upon which the Hotstring function will operate. If the script is suspended, newly added/enabled hotstrings will also be suspended until the suspension is turned off (unless they are exempt as described in the Suspend section). The keyboard and/or mouse hooks will be installed or removed if justified by the changes made by this function. This function cannot directly enable or disable hotstrings in scripts other than its own. Once a script has at least one hotstring, it becomes persistent, meaning that ExitApp rather than Exit should be used to terminate it. Hotstring scripts are also automatically #SingleInstance unless #SingleInstance Off has been specified. Variant (Duplicate) Hotstrings A particular hotstring can be created more than once if each definition has different HotIf criteria, case-sensitivity (C vs. C0/C1), or word-sensitivity (?). These are known as hotstring variants. For example: HotIfWinActive "ahk_group CarForums" Hotstring "::btw", "behind the wheel" HotIfWinActive "Inter-Office Chat" Hotstring "::btw", "back to work" HotIfWinActive Hotstring "::btw", "by the way" If more than one variant of a hotstring is eligible to fire, only the one created earliest will fire. For more information about IfWin, see #HotIf's General Remarks. EndChars OldValue := Hotstring("EndChars" , NewValue) Retrieves or modifies the set of characters used as ending characters by the hotstring recognizer. For example: prev_chars := Hotstring("EndChars", "-()[]{}':;`"/\,.?!`n `t") MsgBox "The previous value was: " prev_chars #Hotstring EndChars also affects this setting. It is currently not possible to specify a different set of end characters for each hotstring. MouseReset OldValue := Hotstring("MouseReset" , NewValue) Retrieves or modifies the global setting which controls whether mouse clicks reset the hotstring recognizer, as described here. NewValue should be 1 (true) to enable mouse click detection and resetting of the hotstring recognizer, or 0 (false) to disable it. The return value is the setting which was in effect before the function was called. The mouse hook may be installed or removed if justified by the changes made by this function. #Hotstring NoMouse also affects this setting, and is equivalent to specifying false for NewValue. Reset Hotstring "Reset" Immediately resets the hotstring recognizer. In other words, the script will begin waiting for an entirely new hotstring, eliminating from consideration anything you previously typed. Setting Default Options Hotstring NewOptions To set new default options for subsequently created hotstrings, pass the options to the Hotstring function without any leading or trailing colon. For example: Hotstring "T". Turning on case-sensitivity (C) or word-sensitivity (?) also affects which existing hotstrings will be found by any subsequent calls to the Hotstring function. For example, Hotstring ":T:btw" will find ::BTW by default, but not if Hotstring "C" or #Hotstring C is in effect. This can be undone or overridden by passing a mutually-exclusive option; for example, C0 and C1 override C. Related Hotstrings, #HotIf, A_ThisHotkey, #MaxThreadsPerHotkey, Suspend, Threads, Thread, Critical Examples Hotstring Helper. The following script might be useful if you are a heavy user of hotstrings. It's based on the v1 script created by Andreas Borutta. 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. #h:: ; Win+H hotkey { ; Get the text currently selected. The clipboard

is used instead of ; EditGetSelectedText because it works in a greater variety of editors ; (namely word processors). Save the current clipboard contents to be ; restored later. Although this handles only plain text, it seems better ; than nothing: ClipboardOld := A_Clipboard A_Clipboard := "" ; Must start off blank for detection to work. Send "^c" if !ClipWait(1) ; ClipWait timed out. { A_Clipboard := ClipboardOld ; Restore previous contents of clipboard before returning. return } ; Replace CRLF and/or LF with `n for use in a "send-raw" hotstring: ; The same is done for any other characters that might otherwise ; be a problem in raw mode: ClipContent := StrReplace(A_Clipboard, "``", "````") ; Do this replacement first to avoid interfering with the others below. ClipContent := StrReplace (ClipContent, "`r`n", "``n") ClipContent := StrReplace(ClipContent, "`n", "``n") ClipContent := StrReplace(ClipContent, "`t", "``t") ClipContent := StrReplace (ClipContent, ";", "```;") A_Clipboard := ClipboardOld ; Restore previous contents of clipboard. ShowInputBox(":T:`::" ClipContent) } ShowInputBox (DefaultValue) { ; This will move the input box's caret to a more friendly position: SetTimer MoveCaret, 10 ; Show the input box, providing the default hotstring: IB := InputBox(" (Type your abreviation at the indicated insertion point. You can also edit the replacement text if you wish. Example entry: :T:btw`::by the way)", "New Hotstring",, DefaultValue) if IB.Result = "Cancel" ; The user pressed Cancel. return if RegExMatch(IB.Value, "(?P:.*?:(?P.*?))::(?P.*)", &Entered) { if !Entered.Abbreviation MsgText := "You didn't provide an abbreviation" else if !Entered.Replacement MsgText := "You didn't provide a replacement" else { Hotstring Entered.Label, Entered.Replacement ; Enable the hotstring now. FileAppend "`n" IB.Value, A_ScriptFullPath ; Save the hotstring for later use. } } else MsgText := "The hotstring appears to be improperly formatted" if IsSet(MsgText) { Result := MsgBox(MsgText ". Would you like to try again?",, 4) if Result = "Yes" ShowInputBox(DefaultValue) } MoveCaret() { WinWait "New Hotstring" ; Otherwise, move the input box's insertion point to where the user will type the abbreviation. Send "{Home}{Right 3}" SetTimer , 0 } } If - Syntax & Usage | AutoHotkey v2 If Specifies one or more statements to execute if an expression evaluates to true. If Expression { Statements } Remarks If the If statement's expression evaluates to true (which is any result other than an empty string or the number 0), the line or block underneath it is executed. Otherwise, if there is a corresponding Else statement, execution jumps to the line or block underneath it. If an If owns more than one line, those lines must be enclosed in braces (to create a block). However, if only one line belongs to an If, the braces are optional. See the examples at the bottom of this page. The space after if is optional if the expression starts with an open-parenthesis, as in if(expression). The One True Brace (OTB) style may optionally be used. For example: if (x < y) { ; ... } if WinExist("Untitled - Notepad") { WinActivate } if IsDone { ; ... } else { ; ... } Unlike an If statement, an Else statement supports any type of statement immediately to its right. Related Expressions, Ternary operator (a?b:c), Blocks, Else, While-loop Examples If A_Index is greater than 100, return. if (A_Index > 100) return If the result of A_TickCount - StartTime is greater than the result of 2*MaxTime + 100, show "Too much time has passed." and terminate the script. if (A_TickCount - StartTime > 2*MaxTime + 100) { MsgBox "Too much time has passed." ExitApp } This example is executed as follows: If Color is the word "Blue" or "White": Show "The color is one of the allowed values.". Terminate the script. Otherwise if Color is the word "Silver": Show "Silver is not an allowed color.". Stop further checks. Otherwise: Show "This color is not recognized.". Terminate the script. if (Color = "Blue" or Color = "White") { MsgBox "The color is one of the allowed values." ExitApp } else if (Color = "Silver") { MsgBox "Silver is not an allowed color." return } else { MsgBox "This color is not recognized." ExitApp } A single multi-statement line does not need to be enclosed in braces. MyVar := 3 if (MyVar > 2) MyVar++, MyVar := MyVar - 4, MyVar .= " test" MsgBox MyVar ; Reports "0 test". Similar to AutoHotkey v1's If Var [not] between Lower and Upper, the following examples check whether a variable's contents are numerically or alphabetically between two values (inclusive). Checks whether var is in the range 1 to 5: if (var >= 1 and var <= 5) MsgBox var " is in the range 1 to 5, inclusive." Checks whether var is in the range 0.0 to 1.0: if not (var >= 0.0 and var <= 1.0) MsgBox var " is not in the range 0.0 to 1.0, inclusive." Checks whether var is between VarLow and VarHigh (inclusive): if (var >= VarLow and var <= VarHigh) MsgBox var " is between " VarLow " and " VarHigh "." Checks whether var is alphabetically between the words blue and red (inclusive): if (StrCompare(var, "blue") >= 0) and (StrCompare(var, "red") <= 0) MsgBox var " is alphabetically between the words blue and red." Allows the user to enter a number and checks whether it is in the range 1 to 10: LowerLimit := 1 UpperLimit := 10 IB := InputBox("Enter a number between " LowerLimit " and " UpperLimit) if not (IB.Value >= LowerLimit and IB.Value <= UpperLimit) MsgBox "Your input is not within the valid range." Similar to AutoHotkey v1's If Var [not] in/contains MatchList, the following examples check whether a variable's contents match one of the items in a list. Checks whether var is the file extension exe, bat or com: if (var ~= "i)\A(exe|bat|com)\z") MsgBox "The file extension is an executable type." Checks whether var is the prime number 1, 2, 3, 5, 7 or 11: if (var ~= "\A(1|2|3|5|7|11)\z") MsgBox var " is a small prime number." Checks whether var contains the digit 1 or 3: if (var ~= "1|3") MsgBox "Var contains the digit 1 or 3 (Var could be 1, 3, 10, 21, 23, etc.)" Checks whether var is one of the items in MyItemList: ; Uncomment the following line if MyItemList contains RegEx chars except | ; MyItemList := RegExReplace(MyItemList, "[\Q\.*?+[{()^$\E]", "\$0") if (var ~= "i)\A(" MyItemList ")\z") MsgBox var " is in the list." Allows the user to enter a string and checks whether it is the word yes or no: IB := InputBox("Enter YES or NO") if not (IB.Value ~= "i)\A(yes|no)\z") MsgBox "Your input is not valid." Checks whether active_title contains "Address List.txt" or "Customer List.txt" and checks whether it contains "metapad" or "Notepad": active_title := WinGetTitle("A") if (active_title ~= "i)Address List\.txt|Customer List\.txt") MsgBox "One of the desired windows is active." if not (active_title ~= "i)metapad|Notepad") MsgBox "But the file is not open in either Metapad or Notepad." ImageSearch - Syntax & Usage | AutoHotkey v2 ImageSearch Searches a region of the screen for an image. ImageSearch &OutputVarX, &OutputVarY, X1, Y1, X2, Y2, ImageFile Parameters &OutputVarX, &OutputVarY Type: VarRef References to the output variables in which to store the X and Y coordinates of the upper-left pixel of where the image was found on the screen (if no match is found, the variables are made blank). Coordinates are relative to the active window's client area unless CoordMode was used to change that. X1, Y1 Type: Integer The X and Y coordinates of the upper left corner of the rectangle to search. Coordinates are relative to the active window's client area unless CoordMode was used to change that. X2, Y2 Type: Integer The X and Y coordinates of the lower right corner of the rectangle to search. Coordinates are relative to the active window's client area unless CoordMode was used to change that. ImageFile Type: String The file name of an image, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Supported image formats include ANI, BMP, CUR, EMF, Exif, GIF, ICO, JPG, PNG, TIF, and WMF (BMP images must be 16-bit or higher). Other sources of icons include the following types of files: EXE, DLL, CPL, SCR, and other types that contain icon resources. Options: Zero or more of the following strings may be also be present immediately before the name of the file. Separate each option from the next with a single space or tab. For example: "*2 *w100 *h-1 C:\Main Logo.bmp". *IconN: To use an icon group other than the first one in the file, specify *Icon followed immediately by the number of the group. For example, *Icon2 would load the default icon from the second icon group. *n (variation): Specify for n a number between 0 and 255 (inclusive) to indicate the allowed number of shades of variation in either direction for the intensity of the red, green, and blue components of each pixel's color. For example, if *2 is specified and the color of a pixel is 0x444444, any color from 0x424242 to 0x464646 will be considered a match. This parameter is helpful if the coloring of the image varies slightly or if ImageFile uses a format such as GIF or JPG that does not accurately represent an image on the screen. If you specify 255 shades of variation, all colors will match. The default is 0 shades. *TransN: This option makes it easier to find a match by specifying one color within the image that will match any color on the screen. It is most commonly used to find PNG, GIF, and TIF files that have some transparent areas (however, icons do not need this option because their transparency is automatically supported). For GIF files, *TransWhite might be most likely to work. For PNG and TIF files, *TransBlack might be best. Otherwise, specify for N some other color name or RGB value (see the color chart for guidance, or use PixelGetColor in its RGB mode). Examples: *TransBlack, *TransFFFFAA, *Trans0xFFFFAA. *wn and *hn: Width and height to which to scale the image (this width and height also determines which icon to load from a multi-icon .ICO file). If both these options are omitted, icons loaded from ICO, DLL, or EXE files are scaled to the system's default small-icon size, which is usually 16 by 16 (you can force the actual/internal size to be used by specifying *w0 *h0). Images that are not icons are loaded at their actual size. To shrink or enlarge the image while preserving its aspect ratio, specify -1 for one of the dimensions and a positive number for the other. For example, specifying *w200 *h-1 would make the image 200 pixels wide and cause its height to be set automatically. A bitmap or icon handle can be used instead of a filename. For example, "HBITMAP:*" handle. Return Value Type: Integer (boolean) This function returns 1 (true) if the image was found in the specified region, or 0 (false) if it was not found. Error Handling A ValueError is thrown if an invalid parameter was detected or the image could not be loaded. An OSError is thrown if an internal function call fails. Remarks ImageSearch can be used to detect graphical objects on the screen that either lack text or whose text cannot be easily retrieved. For example, it can be used to discover the position of picture buttons, icons, web page links, or game objects. Once located, such objects can be clicked via Click. A strategy that is sometimes useful is to search for a small clipping from an image rather than the entire image. This can improve reliability in cases where the image as a whole varies, but certain parts within it are always the same. One way to extract a clipping is to: Press Alt+PrtSc while the image is visible in the active window. This places a screenshot on the clipboard. Open an image processing program such as Paint. Paste the contents of the clipboard (that is, the screenshot). Select a region that does not vary and that is unique to the image. Copy and paste that region to a new image document. Save it as a small file for use with ImageSearch. To be a match, an image on the screen must be the same size as the one loaded via the ImageFile parameter and its options. The region to be searched must be visible; in other words, it is not possible to search a region of a window hidden behind another window. By contrast, images that lie partially beneath the mouse cursor can usually be detected. The exception to this is game cursors, which in most cases will obstruct any images beneath them. Since the search starts at the top row of the region and moves downward, if there is more than one match, the one closest to the top will be found. Icons containing a transparent color automatically allow that color to match any color on the screen. Therefore, the color of what lies behind the icon does not matter. ImageSearch supports 8-bit color screens (256-color) or higher. The search behavior may vary depending on the display adapter's color depth (especially for GIF and JPG files). Therefore, if a script will run under multiple color depths, it is best to test it on each depth setting. You can use the shades-of-variation option (*n) to help make the behavior consistent across multiple color depths. If the image on the screen is translucent, ImageSearch will probably fail to find it. To work around this, try the shades-of-variation option (*n) or make the window temporarily opaque via WinSetTransparent("Off"). Related PixelSearch, PixelGetColor, CoordMode, MouseGetPos Examples Searches a region of the active window for an image and stores in FoundX and FoundY the X and Y coordinates of the upper-left pixel of where the image was found. ImageSearch

&FoundX, &FoundY, 40, 40, 300, 300, "C:\My Images\test.bmp" Searches a region of the screen for an image and stores in FoundX and FoundY the X and Y coordinates of the upper-left pixel of where the image was found, including advanced error handling. CoordMode "Pixel" ; Interprets the coordinates below as relative to the screen rather than the active window's client area. try { if ImageSearch(&FoundX, &FoundY, 0, 0, A_ScreenWidth, A_ScreenHeight, "*Icon3 " A_ProgramFiles "\SomeApp\SomeApp.exe") MsgBox "The icon was found at " FoundX "x" FoundY else MsgBox "Icon could not be found on the screen." } catch as exc MsgBox "Could not conduct the search due to the following error:`n" exc.Message Alphabetical Function Index | AutoHotkey v2 Alphabetical Function Index Click on a function name for details. Entries in large font are the most commonly used. Go to entries starting with: E, I, M, S, W, # Name Description { ... } (Block) Blocks are one or more statements enclosed in braces. Typically used with function definitions and control flow statements. { ... } / Object Creates an Object from a list of property name and value pairs. [...] / Array Creates an Array from a sequence of parameter values. Abs Returns the absolute value of the specified number. ASin Returns the arcsine (the number whose sine is the specified number) in radians. ACos Returns the arccosine (the number whose cosine is the specified number) in radians. ATan Returns the arctangent (the number whose tangent is the specified number) in radians. BlockInput Disables or enables the user's ability to interact with the computer via keyboard and mouse. Break Exits (terminates) any type of loop statement. Buffer Creates a Buffer, which encapsulates a block of memory for use with other functions. CallbackCreate Creates a machine-code address that when called, redirects the call to a function in the script. CallbackFree Frees a callback created by CallbackCreate. CaretGetPos Retrieves the current position of the caret (text insertion point). Catch Specifies the code to execute if a value or error is thrown during execution of a try statement. Ceil Returns the specified number rounded up to the nearest integer (without any .00 suffix). Chr Returns the string (usually a single character) corresponding to the character code indicated by the specified number. Click Clicks a mouse button at the specified coordinates. It can also hold down a mouse button, turn the mouse wheel, or move the mouse. ClipboardAll Creates an object containing everything on the clipboard (such as pictures and formatting). ClipWait Waits until the clipboard contains data. ComCall Calls a native COM interface method by index. ComObjActive Retrieves a registered COM object. ComObjArray Creates a SafeArray for use with COM. ComObjConnect Connects a COM object's event source to the script, enabling events to be handled. ComObject Creates a COM object. ComObjFlags Retrieves or changes flags which control a COM wrapper object's behaviour. ComObjFromPtr Wraps a raw IDispatch pointer (COM object) for use by the script. ComObjGet Returns a reference to an object provided by a COM component. ComObjQuery Queries a COM object for an interface or service. ComObjType Retrieves type information from a COM object. ComObjValue Retrieves the value or pointer stored in a COM wrapper object. ComValue Wraps a value, SafeArray or COM object for use by the script or for passing to a COM method. Continue Skips the rest of a loop statement's current iteration and begins a new one. ControlAddItem Adds the specified string as a new entry at the bottom of a ListBox or ComboBox. ControlChooseIndex Sets the selection in a ListBox, ComboBox or Tab control to be the specified entry or tab number. ControlChooseString Sets the selection in a ListBox or ComboBox to be the first entry whose leading part matches the specified string. ControlClick Sends a mouse button or mouse wheel event to a control. ControlDeleteItem Deletes the specified entry number from a ListBox or ComboBox. ControlFindItem Returns the entry number of a ListBox or ComboBox that is a complete match for the specified string. ControlFocus Sets input focus to a given control on a window. ControlGetChecked Returns a non-zero value if the checkbox or radio button is checked. ControlGetChoice Returns the name of the currently selected entry in a ListBox or ComboBox. ControlGetClassNN Returns the ClassNN (class name and sequence number) of the specified control. ControlGetEnabled Returns a non-zero value if the specified control is enabled. ControlGetFocus Retrieves which control of the target window has keyboard focus, if any. ControlGetHwnd Returns the unique ID number of the specified control. ControlGetIndex Returns the index of the currently selected entry or tab in a ListBox, ComboBox or Tab control. ControlGetItems Returns an array of items/rows from a ListBox, ComboBox, or DropDownList. ControlGetPos Retrieves the position and size of a control. ControlGetStyleControlGetExStyle Returns an integer representing the style or extended style of the specified control. ControlGetText Retrieves text from a control. ControlGetVisible Returns a non-zero value if the specified control is visible. ControlHide Hides the specified control. ControlHideDropDown Hides the drop-down list of a ComboBox control. ControlMove Moves or resizes a control. ControlSendControlSendText Sends simulated keystrokes or text to a window or control. ControlSetChecked Turns on (checks) or turns off (unchecks) a checkbox or radio button. ControlSetEnabled Enables or disables the specified control. ControlSetStyleControlSetExStyle Changes the style or extended style of the specified control, respectively. ControlSetText Changes the text of a control. ControlShow Shows the specified control if it was previously hidden. ControlShowDropDown Shows the drop-down list of a ComboBox control. CoordMode Sets coordinate mode for various built-in functions to be relative to either the active window or the screen. Cos Returns the trigonometric cosine of the specified number. Critical Prevents the current thread from being interrupted by other threads, or enables it to be interrupted. DateAdd Adds or subtracts time from a date-time value. DateDiff Compares two date-time values and returns the difference. DetectHiddenText Determines whether invisible text in a window is "seen" for the purpose of finding the window. This affects built-in functions such as WinExist and WinActivate. DetectHiddenWindows Determines whether invisible windows are "seen" by the script. DirCopy Copies a folder along with all its sub-folders and files (similar to xcopy). DirCreate Creates a folder. DirDelete Deletes a folder. DirExist Checks for the existence of a folder and returns its attributes. DirMove Moves a folder along with all its sub-folders and files. It can also rename a folder. DirSelect Displays a standard dialog that allows the user to select a folder. DllCall Calls a function inside a DLL, such as a standard Windows API function. Download Downloads a file from the Internet. DriveEject Ejects the tray of the specified CD/DVD drive, or ejects a removable drive. DriveGetCapacity Returns the total capacity of the drive which contains the specified path, in megabytes. DriveGetFileSystem Returns the type of the specified drive's file system. DriveGetLabel Returns the volume label of the specified drive. DriveGetList Returns a string of letters, one character for each drive letter in the system. DriveGetSerial Returns the volume serial number of the specified drive. DriveGetSpaceFree Returns the free disk space of the drive which contains the specified path, in megabytes. DriveGetStatus Returns the status of the drive which contains the specified path. DriveGetStatusCD Returns the media status of the specified CD/DVD drive. DriveGetType Returns the type of the drive which contains the specified path. DriveLock Prevents the eject feature of the specified drive from working. DriveRetract Retracts the tray of the specified CD/DVD drive. DriveSetLabel Changes the volume label of the specified drive. DriveUnlock Restores the eject feature of the specified drive. Edit Opens the current script for editing in the associated editor. EditGetCurrentCol Returns the column number in an Edit control where the caret (text insertion point) resides. EditGetCurrentLine Returns the line number in an Edit control where the caret (text insert point) resides. EditGetLine Returns the text of the specified line in an Edit control. EditGetLineCount Returns the number of lines in an Edit control. EditGetSelectedText Returns the selected text in an Edit control. EditPaste Pastes the specified string at the caret (text insertion point) in an Edit control. Else Specifies one or more statements to execute if the associated statement's body did not execute. EnvGet Retrieves an environment variable. EnvSet Writes a value to a variable contained in the environment. Exit Exits the current thread. ExitApp Terminates the script. Exp Returns e (which is approximately 2.71828182845905) raised to the Nth power. FileAppend Writes text or binary data to the end of a file (first creating the file, if necessary). FileCopy Copies one or more files. FileCreateShortcut Creates a shortcut (.lnk) file. FileDelete Deletes one or more files. FileEncoding Sets the default encoding for FileRead, Loop Read, FileAppend, and FileOpen. FileExist Checks for the existence of a file or folder and returns its attributes. FileInstall Includes the specified file inside the compiled version of the script. FileGetAttrib Reports whether a file or folder is read-only, hidden, etc. FileGetShortcut Retrieves information about a shortcut (.lnk) file, such as its target file. FileGetSize Retrieves the size of a file. FileGetTime Retrieves the datetime stamp of a file or folder. FileGetVersion Retrieves the version of a file. FileMove Moves or renames one or more files. FileOpen Opens a file to read specific content from it and/or to write new content into it. FileRead Retrieves the contents of a file. FileRecycle Sends a file or directory to the recycle bin if possible, or permanently deletes it. FileRecycleEmpty Empties the recycle bin. FileSelect Displays a standard dialog that allows the user to open or save file(s). FileSetAttrib Changes the attributes of one or more files or folders. Wildcards are supported. FileSetTime Changes the datetime stamp of one or more files or folders. Wildcards are supported. Finally Ensures that one or more statements are always executed after a Try statement finishes. Float Converts a numeric string or integer value to a floating-point number. Floor Returns the specified number rounded down to the nearest integer (without any .00 suffix). For Repeats a series of functions once for each key-value pair in an object. Format Formats a variable number of input values according to a format string. FormatTime Transforms a YYYYMMDDHH24MISS timestamp into the specified date/time format. GetKeyName Retrieves the name or text of a key. GetKeyVK Retrieves the virtual key code of a key. GetKeySC Retrieves the scan code of a key. GetKeyState Checks if a keyboard key or mouse/joystick button is down or up. Also retrieves joystick status. GetMethod Retrieves the implementation function of a method. Goto Jumps to the specified label and continues execution. GroupActivate Activates the next window in a window group that was defined with GroupAdd. GroupAdd Adds a window specification to a window group, creating the group if necessary. GroupClose Closes the active window if it was just activated by GroupActivate or GroupDeactivate. It then activates the next window in the series. It can also close all windows in a group. GroupDeactivate Similar to GroupActivate except activates the next window not in the group. Gui() Creates and returns a new Gui object. This can be used to define a custom window, or graphical user interface (GUI), to display information or accept user input. GuiCtrlFromHwnd Retrieves the GuiControl object of a GUI control associated with the specified HWND. GuiFromHwnd Retrieves the Gui object of a GUI window associated with the specified HWND. HasBase Returns a non-zero number if the specified value is derived from the specified base object. HasMethod Returns a non-zero number if the specified value has a method by the specified name. HasProp Returns a non-zero number if the specified value has a property by the specified name. HotIf / HotIfWin... Specifies the criteria for subsequently created or modified hotkey variants. Hotkey Creates, modifies, enables, or disables a hotkey while the script is running. Hotstring Creates, modifies, enables, or disables a hotstring while the script is running. If Specifies one or more statements to execute if an expression evaluates to true. IL_Create IL_Add IL_Destroy The means by which icons are added to a ListView or TreeView control. ImageSearch Searches a region of the screen for an image. IniDelete Deletes a value from a standard format .ini file. IniRead Reads a value, section or list of section names from a standard format .ini file. IniWrite Writes a value or section to a standard format .ini file. InputBox Displays an input box to ask the user to enter a string. InputHook Creates an object which can be used to collect or intercept keyboard input. InstallKeybdHook Installs or uninstalls the keyboard hook.

InstallMouseHook Installs or uninstalls the mouse hook. InStr Searches for a given occurrence of a string, from the left or the right. Integer Converts a numeric string or floating-point value to an integer. IsLabel Returns a non-zero number if the specified label exists in the current scope. IsObject Returns a non-zero number if the specified value is an object. IsSet / IsSetRef Returns a non-zero number if the specified variable has been assigned a value. KeyHistory Displays script info and a history of the most recent keystrokes and mouse clicks. KeyWait Waits for a key or mouse/joystick button to be released or pressed down. ListHotkeys Displays the hotkeys in use by the current script, whether their subroutines are currently running, and whether or not they use the keyboard or mouse hook. ListLines Enables or disables line logging or displays the script lines most recently executed. ListVars Displays the script's variables: their names and current contents. ListViewGetContent Returns a list of items/rows from a ListView. LoadPicture Loads a picture from file and returns a bitmap or icon handle. Log Returns the logarithm (base 10) of the specified number. Ln Returns the natural logarithm (base e) of the specified number. Loop (normal) Performs a series of functions repeatedly: either the specified number of times or until break is encountered. 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. Loop Reg Retrieves the contents of the specified registry subkey, one item at a time. Map Creates a Map from a list of key-value pairs. Max Returns the highest value of one or more numbers. MenuBar() Creates a MenuBar object, which can be used to define a GUI menu bar. Menu() Creates a Menu object, which can be used to create and display a menu. MenuFromHandle Retrieves the Menu or MenuBar object corresponding to a Win32 menu handle. MenuSelect Invokes a menu item from the menu bar of the specified window. Min Returns the lowest value of one or more numbers. Mod Modulo. Returns the remainder of the specified dividend divided by the specified divisor. MonitorGet Checks if the specified monitor exists and optionally retrieves its bounding coordinates. MonitorGetCount Returns the total number of monitors. MonitorGetName Returns the operating system's name of the specified monitor. MonitorGetPrimary Returns the number of the primary monitor. MonitorGetWorkArea Checks if the specified monitor exists and optionally retrieves the bounding coordinates of its working area. MouseClick Clicks or holds down a mouse button, or turns the mouse wheel. NOTE: The Click function is generally more flexible and easier to use. MouseClickDrag Clicks and holds the specified mouse button, moves the mouse to the destination coordinates, then releases the button. MouseGetPos Retrieves the current position of the mouse cursor, and optionally which window and control it is hovering over. MouseMove Moves the mouse cursor. MsgBox Displays the specified text in a small window containing one or more buttons (such as Yes and No). Number Converts a numeric string to a pure integer or floating-point number. NumGet Returns the binary number stored at the specified address+offset. NumPut Stores one or more numbers in binary format at the specified address+offset. ObjAddRef / ObjRelease Increments or decrements an object's reference count. ObjBindMethod Creates a BoundFunc object which calls a method of a given object. ObjHasOwnProp ObjOwnProps These functions are equivalent to built-in methods of the Object type. It is usually recommended to use the corresponding method instead. ObjGetBase Retrieves an object's base object. ObjGetCapacity Returns the current capacity of the object's internal array of properties. ObjOwnPropCount Returns the number of properties owned by an object. ObjSetBase Sets an object's base object. ObjSetCapacity Sets the current capacity of the object's internal array of own properties. OnClipboardChange Causes the specified function to be called automatically whenever the clipboard's content changes. OnError Causes the specified function to be called automatically when an unhandled error occurs. OnExit Causes the specified function to be called automatically when the script exits. OnMessage Causes the specified function to be called automatically whenever the script receives the specified message. Ord Returns the ordinal value (numeric character code) of the first character in the specified string. OutputDebug Sends a string to the debugger (if any) for display. Pause Pauses the script's current thread. Persistent Prevents the script from exiting automatically when its last thread completes, allowing it to stay running in an idle state. PixelGetColor Retrieves the color of the pixel at the specified x,y coordinates. PixelSearch Searches a region of the screen for a pixel of the specified color. PostMessage Places a message in the message queue of a window or control. ProcessClose Forces the first matching process to close. ProcessExist Checks if the specified process exists. ProcessGetName Returns the name of the specified process. ProcessGetParent Returns the process ID (PID) of the process which created the specified process. ProcessGetPath Returns the path of the specified process. ProcessSetPriority Changes the priority level of the first matching process. ProcessWait Waits for the specified process to exist. ProcessWaitClose Waits for all matching processes to close. Random Generates a pseudo-random number. RegExMatch Determines whether a string contains a pattern (regular expression). RegExReplace Replaces occurrences of a pattern (regular expression) inside a string. RegCreateKey Creates a registry key without writing a value. RegDelete Deletes a value from the registry. RegDeleteKey Deletes a subkey from the registry. RegRead Reads a value from the registry. RegWrite Writes a value to the registry. Reload Replaces the currently running instance of the script with a new one. Return Returns from a subroutine to which execution had previously jumped via function-call, Hotkey activation, or other means. Round Returns the specified number rounded to N decimal places. Run Runs an external program. RunAs Specifies a set of user credentials to use for all subsequent uses of Run and RunWait. RunWait Runs an external program and waits until it finishes. Send / SendText / SendInput / SendPlay / SendEvent Sends simulated keystrokes and mouse clicks to the active window. SendLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. SendMessage Sends a message to a window or control and waits for acknowledgement. SendMode Makes Send synonymous with SendEvent or SendPlay rather than the default (SendInput). Also makes Click and MouseMove/Click/Drag use the specified method. SetCapsLockState Sets the state of CapsLock. Can also force the key to stay on or off. SetControlDelay Sets the delay that will occur after each control-modifying function. SetDefaultMouseSpeed Sets the mouse speed that will be used if unspecified in Click and MouseMove/Click/Drag. SetKeyDelay Sets the delay that will occur after each keystroke sent by Send or ControlSend. SetMouseDelay Sets the delay that will occur after each mouse movement or click. SetNumLockState Sets the state of NumLock. Can also force the key to stay on or off. SetScrollLockState Sets the state of ScrollLock. Can also force the key to stay on or off. SetRegView Sets the registry view used by RegRead, RegWrite, RegDelete, RegDeleteKey and Loop Reg, allowing them in a 32-bit script to access the 64-bit registry view and vice versa. SetStoreCapsLockMode Whether to restore the state of CapsLock after a Send. SetTimer Causes a function to be called automatically and repeatedly at a specified time interval. SetTitleMatchMode Sets the matching behavior of the WinTitle parameter in built-in functions such as WinWait. SetWinDelay Sets the delay that will occur after each windowing function, such as WinActivate. SetWorkingDir Changes the script's current working directory. Shutdown Shuts down, restarts, or logs off the system. Sin Returns the trigonometric sine of the specified number. Sleep Waits the specified amount of time before continuing. Sort Arranges a variable's contents in alphabetical, numerical, or random order (optionally removing duplicates). SoundBeep Emits a tone from the PC speaker. SoundGetInterface Retrieves a native COM interface of a sound device or component. SoundGetMute Retrieves a mute setting of a sound device. SoundGetName Retrieves the name of a sound device or component. SoundGetVolume Retrieves a volume setting of a sound device. SoundPlay Plays a sound, video, or other supported file type. SoundSetMute Changes a mute setting of a sound device. SoundSetVolume Changes a volume setting of a sound device. SplitPath Separates a file name or URL into its name, directory, extension, and drive. Sqrt Returns the square root of the specified number. StatusBarGetText Retrieves the text from a standard status bar control. StatusBarWait Waits until a window's status bar contains the specified string. StrCompare Compares two strings alphabetically. StrGet Copies a string from a memory address or buffer, optionally converting it from a given code page. String Converts a value to a string. StrLen Retrieves the count of how many characters are in a string. StrLower Converts a string to lowercase. StrPtr Returns the current memory address of a string. StrPut Copies a string to a memory address or buffer, optionally converting it to a given code page. StrReplace Replaces the specified substring with a new string. StrSplit Separates a string into an array of substrings using the specified delimiters. StrUpper Converts a string to uppercase. SubStr Retrieves one or more characters from the specified position in a string. Suspend Disables or enables all or selected hotkeys and hotstrings. Switch Executes one case from a list of mutually exclusive candidates. SysGet Retrieves dimensions of system objects, and other system properties. SysGetIPAddresses Returns an array of the system's IPv4 addresses. Tan Returns the trigonometric tangent of the specified number. Thread Sets the priority or interruptibility of threads. It can also temporarily disable all timers. Throw Signals the occurrence of an error. This signal can be caught by a try-catch statement. ToolTip Creates an always-on-top window anywhere on the screen. TraySetIcon Changes the script's tray icon (which is also used by GUI and dialog windows). TrayTip Creates a balloon message window near the tray icon. On Windows 10, a toast notification may be shown instead. Trim / LTrim / RTrim Trims characters from the beginning and/or end of a string. Try Guards one or more statements against runtime errors and values thrown by the throw statement. Type Returns the class name of a value. Until Applies a condition to the continuation of a Loop or For-loop. VarSetStrCapacity Enlarges a variable's holding capacity or frees its memory. This is not normally needed, but may be used with DllCall or SendMessage or to optimize repeated concatenation. VerCompare Compares two version strings. While-loop Performs a series of functions repeatedly until the specified expression evaluates to false. WinActivate Activates the specified window. WinActivateBottom Same as WinActivate except that it activates the bottommost matching window rather than the topmost. WinActive Checks if the specified window is active and returns its unique ID (HWND). WinClose Closes the specified window. WinExist Checks if the specified window exists and returns the unique ID (HWND) of the first matching window. WinGetClass Retrieves the specified window's class name. WinGetClientPos Retrieves the position and size of the specified window's client area. WinGetControls Returns the control names for all controls in the specified window. WinGetControlsHwnd Returns the unique ID numbers for all controls in the specified window. WinGetCount Returns the number of existing windows that match the specified criteria. WinGetID Returns the unique ID number of the specified window. WinGetIDLast Returns the unique ID number of the last/bottommost window if there is more than one match. WinGetList Returns the unique ID numbers of all existing windows that match the specified criteria. WinGetMinMax Returns the state whether the specified window is maximized or minimized. WinGetPID Returns the Process ID number of the specified window. WinGetPos Retrieves the position and size of the specified window. WinGetProcessName Returns the name of the process that owns the specified window. WinGetProcessPath Returns the full path and name of the process that owns the specified window. WinGetStyleWinGetExStyle Returns the style or extended style (respectively) of the specified window. WinGetText Retrieves the text from the specified window. WinGetTitle Retrieves the title of the specified window. WinGetTransColor Returns the color that is marked transparent in the specified window. WinGetTransparent Returns the degree of transparency of the specified window. WinHide Hides the specified window. WinKill Forces the specified window to close. WinMaximize Enlarges the specified window to its maximum

size. WinMinimize Collapses the specified window into a button on the task bar. WinMinimizeAll / WinMinimizeAllUndo Minimizes or unminimizes all windows. WinMove Changes the position and/or size of the specified window. WinMoveBottom Sends the specified window to the bottom of stack; that is, beneath all other windows. WinMoveTop Brings the specified window to the top of the stack without explicitly activating it. WinRedraw Redraws the specified window. WinRestore Unminimizes or unmaximizes the specified window if it is minimized or maximized. WinSetAlwaysOnTop Makes the specified window stay on top of all other windows (except other always-on-top windows). WinSetEnabled Enables or disables the specified window. WinSetRegion Changes the shape of the specified window to be the specified rectangle, ellipse, or polygon. WinSetStyleWinSetExStyle Changes the style or extended style of the specified window, respectively. WinSetTitle Changes the title of the specified window. WinSetTransColor Makes all pixels of the chosen color invisible inside the specified window. WinSetTransparent Makes the specified window semi-transparent. WinShow Unhides the specified window. WinWait Waits until the specified window exists. WinWaitActive / WinWaitNotActive Waits until the specified window is active or not active. WinWaitClose Waits until no matching windows can be found. #ClipboardTimeout Changes how long the script keeps trying to access the clipboard when the first attempt fails. #DllLoad Loads a DLL or EXE file before the script starts executing. #ErrorStdOut Sends any syntax error that prevents a script from launching to the standard error stream (stderr) rather than displaying a dialog. #Hotstring Changes hotstring options or ending characters. #HotIf Creates context-sensitive hotkeys and hotstrings. Such hotkeys perform a different action (or none at all) depending on any condition (an expression). #HotIfTimeout Sets the maximum time that may be spent evaluating a single #HotIf expression. #Include / #IncludeAgain Causes the script to behave as though the specified file's contents are present at this exact position. #InputLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. #MaxThreads Sets the maximum number of simultaneous threads. #MaxThreadsBuffer Causes some or all hotkeys to buffer rather than ignore keypresses when their #MaxThreadsPerHotkey limit has been reached. #MaxThreadsPerHotkey Sets the maximum number of simultaneous threads per hotkey or hotstring. #NoTrayIcon Disables the showing of a tray icon. #Requires Displays an error and quits if a version requirement is not met. #SingleInstance Determines whether a script is allowed to run again when it is already running. #SuspendExempt Exempts subsequent hotkeys and hotstrings from suspension. #UseHook Forces the use of the hook to implement all or some keyboard hotkeys. #Warn Enables or disables warnings for specific conditions which may indicate an error, such as a typo or missing "global" declaration. #WinActivateForce Skips the gentle method of activating a window and goes straight to the forceful method. IniDelete - Syntax & Usage | AutoHotkey v2 IniDelete Deletes a value from a standard format .ini file. IniDelete Filename, Section , Key Parameters Filename Type: String The name of the .ini file, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Section Type: String The section name in the .ini file, which is the heading phrase that appears in square brackets (do not include the brackets in this parameter). Key Type: String The key name in the .ini file. If omitted, the entire Section will be deleted. Error Handling An OSError is thrown on failure. Regardless of whether an exception is thrown, A_LastError is set to the result of the operating system's GetLastError() function. Remarks A standard ini file looks like: [SectionName] Key=Value Related IniRead, IniWrite, RegDelete, RegDeleteKey Examples Deletes a key and its value located in section2 from a standard format .ini file. IniDelete "C:\Temp\myfile.ini", "section2", "key" IniRead - Syntax & Usage | AutoHotkey v2 IniRead Reads a value, section or list of section names from a standard format .ini file. Value := IniRead(Filename, Section, Key , Default) Section := IniRead(Filename, Section) SectionNames := IniRead(Filename) Parameters Filename Type: String The name of the .ini file, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Section Type: String The section name in the .ini file, which is the heading phrase that appears in square brackets (do not include the brackets in this parameter). Key Type: String The key name in the .ini file. Default Type: String The value to return on failure, such as if the requested key, section or file is not found. If omitted, an OSError is thrown on failure. Return Value Type: String This function returns the actual value of the specified key. If the value cannot be retrieved, the default value indicated by the Default parameter is returned. If the Key parameter is omitted, this function returns an entire section. Comments and empty lines are omitted. Only the first 65,533 characters of the section are retrieved. If the Key and Section parameters are omitted, this function returns a linefeed (`n) delimited list of section names. Error Handling An OSError is thrown on failure, but only if Default is omitted. Regardless of whether an exception is thrown, A_LastError is set to the result of the operating system's GetLastError() function. Remarks The operating system automatically omits leading and trailing spaces/tabs from the retrieved string. To prevent this, enclose the string in single or double quote marks. The outermost set of single or double quote marks is also omitted, but any spaces inside the quote marks are preserved. Values longer than 65,535 characters are likely to yield inconsistent results. A standard ini file looks like: [SectionName] Key=Value Unicode: IniRead and IniWrite rely on the external functions GetPrivateProfileString and WritePrivateProfileString to read and write values. These functions support Unicode only in UTF-16 files; all other files are assumed to use the system's default ANSI code page. Related IniDelete, IniWrite, RegRead, file-reading loop, FileRead Examples Reads the value of a key located in section2 from a standard format .ini file and stores it in Value. Value := IniRead("C:\Temp\myfile.ini", "section2", "key") MsgBox "The value is " Value IniWrite - Syntax & Usage | AutoHotkey v2 IniWrite Writes a value or section to a standard format .ini file. IniWrite Value, Filename, Section, Key IniWrite Pairs, Filename, Section Parameters Value Type: String The string or number that will be written to the right of Key's equal sign (=). If the text is long, it can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. Pairs Type: String The complete content of a section to write to the .ini file, excluding the [SectionName] header. Key must be omitted. Pairs must not contain any blank lines. If the section already exists, everything up to the last key=value pair is overwritten. Pairs can contain lines without an equal sign (=), but this may produce inconsistent results. Comments can be written to the file but are stripped out when they are read back by IniRead. Filename Type: String The name of the .ini file, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Section Type: String The section name in the .ini file, which is the heading phrase that appears in square brackets (do not include the brackets in this parameter). Key Type: String The key name in the .ini file. Error Handling An OSError is thrown on failure. Regardless of whether an exception is thrown, A_LastError is set to the result of the operating system's GetLastError() function. Remarks Values longer than 65,535 characters can be written to the file, but may produce inconsistent results as they usually cannot be read correctly by IniRead or other applications. A standard ini file looks like: [SectionName] Key=Value New files are created with a UTF-16 byte order mark to ensure that the full range of Unicode characters can be used. If this is undesired, ensure the file exists before calling IniWrite. For example: ; Create a file with ANSI encoding. FileAppend "", "NonUnicode.ini", "CP0" ; Create a UTF-16 file without byte order mark. FileAppend "[SectionName]`n", "Unicode.ini", "UTF-16-RAW" Unicode: IniRead and IniWrite rely on the external functions GetPrivateProfileString and WritePrivateProfileString to read and write values. These functions support Unicode only in UTF-16 files; all other files are assumed to use the system's default ANSI code page. Related IniDelete, IniRead, RegWrite Examples Writes a value to a key located in section2 of a standard format .ini file. IniWrite "this is a new value", "C:\Temp\myfile.ini", "section2", "key" InputBox - Syntax & Usage | AutoHotkey v2 InputBox Displays an input box to ask the user to enter a string. Obj := InputBox(Prompt, Title, Options, Default) Parameters Prompt Type: String The text of the input box, which is usually a message to the user indicating what kind of input is expected. If Prompt is long, it can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. Title Type: String The title of the input box. If omitted, it defaults to the current value of A_ScriptName. Options Type: String A string of case-insensitive options, with each separated from the last by a space or tab. Xn Yn: The X and Y coordinates of the dialog. For example, X0 Y0 puts the window at the upper left corner of the desktop. If either coordinate is omitted, the dialog will be centered in that dimension. Either coordinate can be negative to position the dialog partially or entirely off the desktop (or on a secondary monitor in a multi-monitor setup). Wn Hn: The width and height of the dialog's client area, which excludes the title bar and borders. For example, W200 H100. T: Specifies the timeout in seconds. For example, T10.0 is ten seconds. If this value exceeds 2147483 (24.8 days), it will be set to 2147483. After the timeout has elapsed, the input box will be automatically closed and Result will be set to word "Timeout". Value will still contain what the user entered. Password: Mask the user's input. To specify which character is used, follow this example: Password* Default Type: String A string that will appear in the input box's edit field when the dialog first appears. The user can change it by backspacing or other means. Return Value Type: Object This function returns an object with the following properties: Value (String): The text entered by the user. Result (String): One of the following words indicating how the input box was closed: "OK", "Cancel", "Timeout". Remarks An input box usually looks like this: The dialog allows the user to enter text and then press OK or CANCEL. The user can resize the dialog window by dragging its borders. A GUI window may display a modal input box by means of OwnDialogs option. A modal input box prevents the user from interacting with the GUI window until the input box is dismissed. Related Gui object, MsgBox, FileSelect, DirSelect, ToolTip, InputHook Examples Allows the user to enter a hidden password. password := InputBox("(your input will be hidden)", "Enter Password", "password").value Allows the user to enter a phone number. IB := InputBox("Please enter a phone number.", "Phone Number", "w640 h480") if IB.Result = "Cancel" MsgBox "You entered '" IB.Value "' but then cancelled." else MsgBox "You entered '" IB.Value "'." InputHook - Syntax & Usage | AutoHotkey v2 InputHook Creates an object which can be used to collect or intercept keyboard input. InputHook := InputHook(Options, EndKeys, MatchList) Parameters Options Type: String A string of zero or more of the following letters (in any order, with optional spaces in between): B: Sets BackspaceIsUndo to false, which causes Backspace to be ignored. C: Sets CaseSensitive to true, making MatchList case sensitive. I: Sets MinSendLevel to 1 or a given value, causing any input with send level below this value to be ignored. For example, I2 would ignore any input with a level of 0 (the default) or 1, but would capture input at level 2. L: Length limit (e.g. L5). The maximum allowed length of the input. When the text reaches this length, the Input is terminated and EndReason is set to the word Max (unless the text matches one of the MatchList phrases, in which case EndReason is set to the word Match). If unspecified, the length limit is 1023. Specifying L0 disables collection of text and the length limit, but does not affect which keys are counted as producing text (see VisibleText). This can be useful in combination with OnChar, OnKeyDown, KeyOpt or EndKeys. M: Modified keystrokes such as Ctrl+A through Ctrl+Z are recognized and transcribed if they correspond to real ASCII characters. Consider this example, which recognizes Ctrl+C: CtrlC := Chr(3) ; Store the character for Ctrl-C in the CtrlC var. ih := InputHook("L1 M") ih.Start() ih.Wait() if (ih.Input = CtrlC) MsgBox "You pressed Control-C." Note: The characters Ctrl+A through Ctrl+Z correspond to Chr(1) through Chr(26). Also, the M option might cause some keyboard shortcuts such as Ctrl+? to misbehave while an Input is in progress. T: Sets

Timeout (e.g. T3 or T2.5). V: Sets VisibleText and VisibleNonText to true. Normally, the user's input is blocked (hidden from the system). Use this option to have the user's keystrokes sent to the active window. *: Wildcard. Sets FindAnywhere to true, allowing matches to be found anywhere within what the user types. E: Handle single-character end keys by character code instead of by keycode. This provides more consistent results if the active window's keyboard layout is different to the script's keyboard layout. It also prevents key combinations which don't actually produce the given end characters from ending input; for example, if @ is an end key, on the US layout Shift+2 will trigger it but Ctrl+Shift+2 will not (if the E option is used). If the C option is also used, the end character is case-sensitive. EndKeys Type: String A list of zero or more keys, any one of which terminates the Input when pressed (the end key itself is not written to the Input buffer). When an Input is terminated this way, EndReason is set to the word EndKey and the EndKey property is set to the name of the key. The EndKeys list uses a format similar to the Send function. For example, specifying {Enter}.{Esc} would cause either Enter, ., or Esc to terminate the Input. To use the braces themselves as end keys, specify {{} and/or {}}. To use Ctrl, Alt, or Shift as end-keys, specify the left and/or right version of the key, not the neutral version. For example, specify {LControl}{RControl} rather than {Control}. Although modified keys such as Alt+C (!c) are not supported, non-alphanumeric characters such as ?!:@&{} by default require the Shift key to be pressed or not pressed depending on how the character is normally typed. If the E option is present, single character key names are interpreted as characters instead, and in those cases the modifier keys must be in the correct state to produce that character. When the E and M options are both used, Ctrl+A through Ctrl+Z are supported by including the corresponding ASCII control characters in EndKeys. An explicit key code such as {vkFF} or {sc001} may also be specified. This is useful in the rare case where a key has no name and produces no visible character when pressed. Its virtual key code can be determined by following the steps at the bottom fo the key list page. MatchList Type: String A comma-separated list of key phrases, any of which will cause the Input to be terminated (in which case EndReason will be set to the word Match). The entirety of what the user types must exactly match one of the phrases for a match to occur (unless the * option is present). In addition, any spaces or tabs around the delimiting commas are significant, meaning that they are part of the match string. For example, if MatchList is ABC , XYZ, the user must type a space after ABC or before XYZ to cause a match. Two consecutive commas results in a single literal comma. For example, the following would produce a single literal comma at the end of string: string1,,,string2. Similarly, the following list contains only a single item with a literal comma inside it: single,,item. Because the items in MatchList are not treated as individual parameters, the list can be contained entirely within a variable. For example, MatchList might consist of List1 "," List2 "," List3 -- where each of the variables contains a large sub-list of match phrases. Input Stack Any number of InputHook objects can be created and in progress at any time, but the order in which they are started affects how input is collected. When each Input is started (by the Start method), it is pushed onto the top of a stack, and is removed from this stack only when the Input is terminated. Keyboard events are passed to each Input in order of most recently started to least. If an Input suppresses a given keyboard event, it is passed no further down the stack. Sent keystrokes are ignored if the send level of the keystroke is below the InputHook's MinSendLevel. In such cases, the keystroke may still be processed by an Input lower on the stack. Multiple InputHooks can be used in combination with MinSendLevel to separately collect both sent keystrokes and real ones. InputHook Object The InputHook function returns an InputHook object, which has the following methods and properties. Methods: KeyOpt: Sets options for a key or list of keys. Start: Starts collecting input. Stop: Terminates the Input and sets EndReason to the word Stopped. Wait: Waits until the Input is terminated (InProgress is false). General Properties: EndKey: Returns the name of the end key which was pressed to terminate the Input. EndMods: Returns a string of the modifiers which were logically down when Input was terminated. EndReason: Returns an EndReason string indicating how Input was terminated. InProgress: Returns true if the Input is in progress and false otherwise. Input: Returns any text collected since the last time Input was started. Match: Returns the MatchList item which caused the Input to terminate. OnEnd: Retrieves or sets the function object which is called when Input is terminated. OnChar: Retrieves or sets the function object which is called after a character is added to the input buffer. OnKeyDown: Retrieves or sets the function object which is called when a notification-enabled key is pressed. OnKeyUp: Retrieves or sets the function object which is called when a notification-enabled key is released. Option Properties: BackspaceIsUndo: Controls whether Backspace removes the most recently pressed character from the end of the Input buffer. CaseSensitive: Controls whether MatchList is case sensitive. FindAnywhere: Controls whether each match can be a substring of the input text. MinSendLevel: Retrieves or sets the minimum send level of input to collect. NotifyNonText: Controls whether the OnKeyDown and OnKeyUp callbacks are called whenever a non-text key is pressed. Timeout: Retrieves or sets the timeout value in seconds. VisibleNonText: Controls whether keys or key combinations which do not produce text are visible (not blocked). VisibleText: Controls whether keys or key combinations which produce text are visible (not blocked). Methods KeyOpt Sets options for a key or list of keys. InputHook.KeyOpt(Keys, KeyOptions) Keys Type: String A list of keys. Braces are used to enclose key names, virtual key codes or scan codes, similar to the Send function. For example, {Enter}.{{} would apply to Enter, . and {. Specifying a key by name, by {vkNN} or by {scNNN} may produce three different results; see below for details. Specify the string {All} (case-insensitive) on its own to apply KeyOptions to all VK and all SC. KeyOpt may then be called a second time to remove options from specific keys. KeyOptions Type: String One or more of the following single-character options (spaces and tabs are ignored). - (minus): Removes any of the options following the -, up to the next +. + (plus): Cancels any previous -, otherwise has no effect. E: End key. If enabled, pressing the key terminates Input, sets EndReason to the word EndKey and the EndKey property to the key's normalized name. Unlike the EndKeys parameter, the state of the Shift key is ignored. For example, @ and 2 are both equivalent to {vk32} on the US keyboard layout. I: Ignore text. Any text normally produced by this key is ignored, and the key is treated as a non-text key (see VisibleNonText). Has no effect if the key normally does not produce text. N: Notify. Causes the OnKeyDown and OnKeyUp callbacks to be called each time the key is pressed. S: Suppresses (blocks) the key after processing it. This overrides VisibleText or VisibleNonText until -S is used. +S implies -V. V: Visible. Prevents the key from being suppressed (blocked). This overrides VisibleText or VisibleNonText until -V is used. +V implies -S. Options can be set by both virtual key code and scan code, and are accumulative. When a key is specified by name, the options are set either by VK or by SC. Where two physical keys share the same VK but differ by SC (such as Up and NumpadUp), they are handled by SC. By contrast, if a VK number is used, it will apply to any physical key which produces that VK (and this may vary over time as it depends on the active keyboard layout). Removing an option by VK number does not affect any options that were set by SC, or vice versa. However, when an option is removed by key name and that name is handled by VK, the option is also removed for the corresponding SC (according to the script's keyboard layout). This allows keys to be excluded by name after applying an option to all keys. If +V is set by VK and +S is set by SC (or vice versa), +V takes precedence. Start Starts collecting input. InputHook.Start() Has no effect if the Input is already in progress. The newly started Input is placed on the top of the InputHook stack, which allows it to override any previously started Input. This method installs the keyboard hook (if it was not already). Stop Terminates the Input and sets EndReason to the word Stopped. InputHook.Stop() Has no effect if the Input is not in progress. Wait Waits until the Input is terminated (InProgress is false). InputHook.Wait(MaxTime) MaxTime Type: Float The maximum number of seconds to wait. If Input is still in progress after MaxTime seconds, the method returns and does not terminate Input. Returns EndReason. General Properties EndKey Returns the name of the end key which was pressed to terminate the Input. KeyName := InputHook.EndKey Note that EndKey returns the "normalized" name of the key regardless of how it was written in EndKeys. For example, {Esc} and {vk1B} both produce Escape. GetKeyName can be used to retrieve the normalized name. If the E option was used, EndKey returns the actual character which was typed (if applicable). Otherwise, the key name is determined according to the script's active keyboard layout. EndKey returns an empty string if EndReason is not "EndKey". EndMods Returns a string of the modifiers which were logically down when Input was terminated. Mods := InputHook.EndMods If all modifiers were logically down (pressed), the full string is: <^>^!<+>+<#># These modifiers have the same meaning as with hotkeys. Each modifier is always qualified with < (left) or > (right). The corresponding key names are: LCtrl, RCtrl, LAlt, RAlt, LShift, RShift, LWin, RWin. InStr can be used to check whether a given modifier (such as >! or ^) is present. The following line can be used to convert Mods to a string of neutral modifiers, such as ^!+#: Mods := RegExReplace (Mods, "[<>](.)(?:>\1)?", "$1") Due to split-second timing, this property may be more reliable than GetKeyState even if it is used immediately after Input terminates, or in the OnEnd callback. EndReason Returns an EndReason string indicating how Input was terminated. Reason := InputHook.EndReason Returns an empty string if the Input is still in progress. InProgress Returns true if the Input is in progress and false otherwise. Boolean := InputHook.InProgress Input Returns any text collected since the last time Input was started. String := InputHook.Input This property can be used while the Input is in progress, or after it has ended. Match Returns the MatchList item which caused the Input to terminate. String := InputHook.Match Returns the matched item with its original case, which may differ from what the user typed if the C option was omitted. Returns an empty string if EndReason is not "Match". OnEnd Retrieves or sets the function object which is called when Input is terminated. MyFunc := InputHook.OnEnd InputHook.OnEnd := MyFunc Type: function object or empty string. Default: empty string. The function is passed one parameter: a reference to the InputHook object. The function is called as a new thread, so starts off fresh with the default values for settings such as SendMode and DetectHiddenWindows. OnChar Retrieves or sets the function object which is called after a character is added to the input buffer. MyFunc := InputHook.OnChar InputHook.OnChar := MyFunc Type: function object or empty string. Default: empty string. The function is passed the following parameters: InputHook, Char. Char is a string containing the character or characters. The presence of multiple characters indicates that a dead key was used prior to the last keypress, but the two keys could not be transliterated to a single character. For example, on some keyboard layouts `e produces Ã¨ while `z produces `z. The function is never called when an end key is pressed. OnKeyDown Retrieves or sets the function object which is called when a notification-enabled key is pressed. MyFunc := InputHook.OnKeyDown InputHook.OnKeyDown := MyFunc Type: function object or empty string. Default: empty string. Key-down notifications must first be enabled by KeyOpt or NotifyNonText. The function is passed the following parameters: InputHook, VK, SC. VK and SC are integers. To retrieve the key name (if any), use GetKeyName(Format("vk{:x}sc{:x}", VK, SC)). The function is called as a new thread, so starts off fresh with the default values for settings such as SendMode and DetectHiddenWindows. The function is never called when an end key is pressed. OnKeyUp Retrieves or sets the function object which is called when a notification-enabled key is released. MyFunc := InputHook.OnKeyUp InputHook.OnKeyUp := MyFunc Type: function object or empty string. Default: empty string. Key-up notifications must first be enabled by KeyOpt or NotifyNonText. Whether a key is considered text or non-text is determined

when the key is pressed. If an InputHook detects a key-up without having detected key-down, it is considered non-text. The function is passed the following parameters: InputHook, VK, SC. VK and SC are integers. To retrieve the key name (if any), use GetKeyName(Format("vk{:x}sc{:x}", VK, SC)). The function is called as a new thread, so starts off fresh with the default values for settings such as SendMode and DetectHiddenWindows. Option Properties BackspaceIsUndo Controls whether Backspace removes the most recently pressed character from the end of the Input buffer. Boolean := InputHook.BackspaceIsUndo InputHook.BackspaceIsUndo := Boolean Type: Integer (boolean). Default: true. Option B sets the value to false. When Backspace acts as undo, it is treated as a text entry key. Specifically, whether the key is suppressed depends on VisibleText rather than VisibleNonText. Backspace is always ignored if pressed in combination with a modifier key such as Ctrl (the logical modifier state is checked rather than the physical state). Note: If the input text is visible (such as in an editor) and the arrow keys or other means are used to navigate within it, Backspace will still remove the last character rather than the one behind the caret (insertion point). CaseSensitive Controls whether MatchList is case sensitive. Boolean := InputHook.CaseSensitive InputHook.CaseSensitive := Boolean Type: Integer (boolean). Default: false. Option C sets the value to true. FindAnywhere Controls whether each match can be a substring of the input text. Boolean := InputHook.FindAnywhere InputHook.FindAnywhere := Boolean Type: Integer (boolean). Default: false. Option * sets the value to true. If true, a match can be found anywhere within what the user types (the match can be a substring of the input text). If false, the entirety of what the user types must match one of the MatchList phrases. In both cases, one of the MatchList phrases must be typed in full. MinSendLevel Retrieves or sets the minimum send level of input to collect. Level := InputHook.MinSendLevel InputHook.MinSendLevel := Level Type: Integer. Default: 0. Option I sets the value to 1 (or a given value). Level should be an integer between 0 and 101. Events which have a send level lower than this value are ignored. For example, a value of 101 causes all input generated by SendEvent to be ignored, while a value of 1 only ignores input at the default send level (zero). The SendInput and SendPlay methods are always ignored, regardless of this setting. Input generated by any source other than AutoHotkey is never ignored as a result of this setting. NotifyNonText Controls whether the OnKeyDown and OnKeyUp callbacks are called whenever a non-text key is pressed. Boolean := InputHook.NotifyNonText InputHook.NotifyNonText := Boolean Type: Integer (boolean). Default: false. Setting this to true enables notifications for all keypresses which do not produce text, such as when pressing Left or Alt+F. Setting this property does not affect a key's options, since the production of text depends on the active window's keyboard layout at the time the key is pressed. NotifyNonText is applied to key-up events by considering whether a previous key-down with a matching VK code was classified as text or non-text. For example, if NotifyNonText is true, pressing Ctrl+A will produce OnKeyDown and OnKeyUp calls for both Ctrl and A, while pressing A on its own will not call OnKeyDown or OnKeyUp unless KeyOpt has been used to enable notifications for that key. See VisibleText for details about which keys are counted as producing text. Timeout Retrieves or sets the timeout value in seconds. Seconds := InputHook.Timeout InputHook.Timeout := Seconds Type: Float. Default: 0.0 (none). Option T also sets the timeout value. The timeout period ordinarily starts when Start is called, but will restart if this property is assigned a value while Input is in progress. If Input is still in progress when the timeout period elapses, it is terminated and EndReason is set to the word Timeout. VisibleNonText Controls whether keys or key combinations which do not produce text are visible (not blocked). Boolean := InputHook.VisibleNonText InputHook.VisibleNonText := Boolean Type: Integer (boolean). Default: true. Option V sets the value to true. If true, keys and key combinations which do not produce text may trigger hotkeys or be passed on to the active window. If false, they are blocked. See VisibleText for details about which keys are counted as producing text. VisibleText Controls whether keys or key combinations which produce text are visible (not blocked). Boolean := InputHook.VisibleText InputHook.VisibleText := Boolean Type: Integer (boolean). Default: false. Option V sets the value to true. If true, keys and key combinations which produce text may trigger hotkeys or be passed on to the active window. If false, they are blocked. Any keystrokes which cause text to be appended to the Input buffer are counted as producing text, even if they do not normally do so in other applications. For instance, Ctrl+A produces text if the M option is used, and Esc produces the control character Chr(27). Dead keys are counted as producing text, although they do not typically produce an immediate effect. Pressing a dead key might also cause the following key to produce text (if only the dead key's character). Backspace is counted as producing text only when it acts as undo. The standard modifier keys and CapsLock, NumLock and ScrollLock are always visible (not blocked). EndReason The EndReason property returns one of the following strings: String Description Stopped The Stop method was called or Start has not yet been called for the first time. Max The Input reached the maximum allowed length and it does not match any of the items in MatchList. Timeout The Input timed out. Match The Input matches one of the items in MatchList. The Match property contains the matched item. EndKey One of the EndKeys was pressed to terminate the Input. The EndKey property contains the terminating key name or character without braces. If the Input is in progress, EndReason is blank. Remarks The Start method must be called before input will be collected. InputHook is designed to allow different parts of the script to monitor input, with minimal conflicts. It can operate continuously, such as to watch for arbitrary words or other patterns. It can also operate temporarily, such as to collect user input or temporarily override specific (or non-specific) keys without interfering with hotkeys. Keyboard hotkeys are still in effect while an Input is in progress, but cannot activate if any of the required modifier keys are suppressed, or if the hotkey uses the reg method and its suffix key is suppressed. For example, the hotkey ^+a:: might be overridden by InputHook, whereas the hotkey $^+a:: would take priority unless the InputHook suppressed Ctrl or Shift. Keys are either suppressed (blocked) or not depending on the following factors (in order): If the V option is in effect for this VK or SC, it is not suppressed. If the S option is in effect for this VK or SC, it is suppressed. If the key is a standard modifier key or CapsLock, NumLock or ScrollLock, it is not suppressed. VisibleText or VisibleNonText is consulted, depending on whether the key produces text. If the property is false, the key is suppressed. See VisibleText for details about which keys are counted as producing text. The keyboard hook is required while an Input is in progress, but will be uninstalled automatically if it is no longer needed when the Input is terminated. The script is automatically persistent while an Input is in progress, so it will continue monitoring input even if there are no running threads. The script may exit automatically when input ends (if there are no running threads and the script is not persistent for some other reason). AutoHotkey does not support Input Method Editors (IME). The keyboard hook intercepts keyboard events and translates them to text by using ToUnicodeEx or ToAsciiEx (except in the case of VK_PACKET events, which encapsulate a single character). If you use multiple languages or keyboard layouts, Input uses the keyboard layout of the active window rather than the script's (regardless of whether the Input is visible). Although not as flexible, hotstrings are generally easier to use. InputHook vs. Input (v1) In AutoHotkey v1.1, InputHook is a replacement for the Input command, offering greater flexbility. The Input command was removed for v2.0, but the code below is mostly equivalent: ; Input OutputVar, % Options, % EndKeys, % MatchList ; v1 ih := InputHook(Options, EndKeys, MatchList) ih.Start() ErrorLevel := ih.Wait() if (ErrorLevel = "EndKey") ErrorLevel .= ":" ih.EndKey OutputVar := ih.Input The Input command terminates any previous Input which it started, whereas InputHook allows more than one Input at a time. Options is interpreted the same, but the default settings differ: The Input command limits the length of the input to 16383, while InputHook limits it to 1023. This can be overridden with the L option, and there is no absolute maximum. The Input command blocks both text and non-text keystrokes by default, and blocks neither if the V option is present. By contrast, InputHook blocks only text keystrokes by default (VisibleNonText defaults to true), so most hotkeys can be used while an Input is in progress. The Input command blocks the thread while it is in progress, whereas InputHook allows the thread to continue, or even exit (which allows any thread that it interrupted to resume). Instead of waiting, the script can register an OnEnd function to be called when the Input is terminated. The Input command returns the user's input only after the Input is terminated, whereas InputHook's Input property allows it to be retrieved at any time. The script can register an OnChar function to be called whenever a character is added, instead of continuously checking the Input property. InputHook gives much more control over individual keys via the KeyOpt method. This includes adding or removing end keys, suppressing or not suppressing specific keys, or ignoring the text produced by specific keys. Unlike the Input command, InputHook can be used to detect keys which do not produce text, without terminating the Input. This is done by registering an OnKeyDown function and using KeyOpt or NotifyNonText to specify which keys are of interest. If a MatchList item caused the Input to terminate, the Match property can be consulted to determine exactly which match (this is more useful when the * option is present). Although the script can consult GetKeyState after the Input command returns, sometimes it does not accurately reflect which keys were pressed when the Input was terminated. InputHook's EndMods property reflects the logical state of the modifier keys at the time Input was terminated. There are some differences relating to backward-compatibility: The Input command stores end keys A-Z in uppercase even though other letters on some keyboard layouts are lowercase. Passing the value to Send would produce a shifted keystroke instead of a plain one. By contrast, InputHook's EndKeys property always returns the normalized name; i.e. whichever character is produced by pressing the key without holding Shift or other modifiers. If a key name used in EndKeys corresponds to a VK which is shared between two physical keys (such as NumpadUp and Up), the Input command handles the primary key by VK and the secondary key by SC, whereas InputHook handles both by SC. {vkNN} notation can be used to handle the key by VK. When the end key is handled by VK, both physical keys can terminate the Input. For example, {NumpadUp} would cause the Input command to be terminated by pressing Up, but ErrorLevel would contain EndKey:NumpadUp since only the VK is considered. When an end key is handled by SC, the Input command always produces names for the known secondary SC of any given VK, and always produces scNNN for any other key (even if it has a name). By contrast, InputHook produces a name if the key has one. Related KeyWait, Hotstrings, InputBox, InstallKeybdHook, Threads Examples Waits for the user to press any single key. MsgBox KeyWaitAny() ; Same again, but don't block the key. MsgBox KeyWaitAny("V") KeyWaitAny(Options:="") { ih := InputHook(Options) if !InStr(Options, "V") ih.VisibleNonText := false ih.KeyOpt("{All}", "E") ; End ih.Start() ih.Wait() return ih.EndKey ; Return the key name } Waits for any key in combination with Ctrl/Alt/Shift/Win. MsgBox KeyWaitCombo() KeyWaitCombo(Options:="") { ih := InputHook(Options) if !InStr(Options, "V") ih.VisibleNonText := false ih.KeyOpt("{All}", "E") ; End ; Exclude the modifiers ih.KeyOpt("{LCtrl}{RCtrl}{LAlt}{RAlt}{LShift}{RShift}{LWin}{RWin}", "-E") ih.Start() ih.Wait() return ih.EndMods . ih.EndKey ; Return a string like <^<+Esc } Simple auto-complete: any day of the week. Pun aside, this is a mostly functional example. Simply run the script and start typing today, press Tab to complete or press Esc to exit. WordList := "Monday`nTuesday`nWednesday`nThursday`nFriday`nSaturday`nSunday" Suffix := "" SacHook := InputHook("V", "{Esc}") SacHook.OnChar := SacChar SacHook.OnKeyDown := SacKeyDown SacHook.KeyOpt("{Backspace}", "N") SacHook.Start() SacChar(ih, char) ; Called when a character is added to

SacHook.Input. { global Suffix := "" if RegExMatch(ih.Input, "`nm)\w+$", &prefix) && RegExMatch(WordList, "`nmi)^" prefix[0] "\K.*", &Suffix) Suffix := Suffix[0] if CaretGetPos(&cx, &cy) ToolTip Suffix, cx + 15, cy else ToolTip Suffix ; Intercept Tab only while we're showing a tooltip. ih.KeyOpt("{Tab}", Suffix = "" ? "-NS" : "+NS") } SacKeyDown(ih, vk, sc) { if (vk = 8) ; Backspace SacChar(ih, "") else if (vk = 9) ; Tab Send "{Text}" Suffix } InstallKeybdHook - Syntax & Usage | AutoHotkey v2 InstallKeybdHook Installs or uninstalls the keyboard hook. InstallKeybdHook Install, Force Parameters Install Type: Boolean Omit this parameter or pass true (any non-zero, non-blank value) to require that the hook be installed. Pass false to remove any requirement previously set by this function and potentially uninstall the hook. Force Type: Boolean If Force is true and Install is omitted or true, the hook is uninstalled and reinstalled. This has the effect of giving it precedence over any hooks previously installed by other processes. If the system has stopped calling the hook due to an unresponsive program, reinstalling the hook might get it working again. If Force is true and Install is false, the hook is uninstalled even if needed for some other purpose. If a hotkey, hotstring or InputHook requires the hook, it will stop working until the hook is reinstalled. The hook may be reinstalled explicitly by calling this function, or automatically as a side-effect of enabling or disabling a hotkey or calling some other function which requires the hook. If Force is false, an internal variable is updated to indicate whether the hook is required by the script, but there might be no immediate change if the hook is required for some other purpose. Remarks The keyboard hook monitors keystrokes for the purpose of activating hotstrings and any keyboard hotkeys not supported by RegisterHotkey (which is a function built into the operating system). It also supports a few other features such as the InputHook function. AutoHotkey does not install the keyboard and mouse hooks unconditionally because together they consume at least 500 KB of memory. Therefore, the keyboard hook is normally installed only when the script contains one of the following: 1) hotstrings; 2) one or more hotkeys that require the keyboard hook (most do not); 3) SetCaps/Scroll/NumLock AlwaysOn/AlwaysOff; 4) active Input hooks. By contrast, the InstallKeybdHook function can be used to unconditionally install the keyboard hook, which has benefits including: KeyHistory can be used to display the last 20 keystrokes (for debugging purposes). The physical state of the modifier keys can be tracked reliably, which removes the need for A_HotkeyModifierTimeout and may improve the reliability with which Send restores the modifier keys to their proper states after temporarily releasing them. GetKeyState can retrieve the physical state of a key. A_TimeIdleKeyboard and A_TimeIdlePhysical can work correctly (ignoring mouse input or artificial input, respectively). Mouse hotkeys which use the Alt modifier (such as !LButton::) can suppress the window menu more efficiently, by sending only one menu mask key when the Alt key is released, instead of sending one each time the button is clicked. Keyboard hotkeys which do not require the hook will use the reg method even if the InstallKeybdHook function is used. By contrast, applying the #UseHook directive or the $ prefix to a keyboard hotkey forces it to require the hook, which causes the hook to be installed if the hotkey is enabled. You can determine whether a script is using the hook via the KeyHistory function or menu item. You can determine which hotkeys are using the hook via the ListHotkeys function or menu item. Related InstallMouseHook, #UseHook, Hotkey, InputHook, KeyHistory, Hotstrings, GetKeyState, KeyWait Examples Installs the keyboard hook unconditionally. InstallKeybdHook InstallMouseHook - Syntax & Usage | AutoHotkey v2 InstallMouseHook Installs or uninstalls the mouse hook. InstallMouseHook Install, Force Parameters Install Type: Boolean Omit this parameter or pass true (any non-zero, non-blank value) to require that the hook be installed. Pass false to remove any requirement previously set by this function and potentially uninstall the hook. Force Type: Boolean If Force is true and Install is omitted or true, the hook is uninstalled and reinstalled. This has the effect of giving it precedence over any hooks previously installed by other processes. If the system has stopped calling the hook due to an unresponsive program, reinstalling the hook might get it working again. If Force is true and Install is false, the hook is uninstalled even if needed for some other purpose. If a hotkey, hotstring or InputHook requires the hook, it will stop working until the hook is reinstalled. The hook may be reinstalled explicitly by calling this function, or automatically as a side-effect of enabling or disabling a hotkey or calling some other function which requires the hook. If Force is false, an internal variable is updated to indicate whether the hook is required by the script, but there might be no immediate change if the hook is required for some other purpose. Remarks The mouse hook monitors mouse clicks for the purpose of activating mouse hotkeys and facilitating hotstrings. AutoHotkey does not install the keyboard and mouse hooks unconditionally because together they consume at least 500 KB of memory (but if the keyboard hook is installed, installing the mouse hook only requires about 50 KB of additional memory; and vice versa). Therefore, the mouse hook is normally installed only when the script contains one or more mouse hotkeys. It is also installed for hotstrings, but that can be disabled via #Hotstring NoMouse. By contrast, the InstallMouseHook function can be used to unconditionally install the mouse hook, which has benefits including: KeyHistory can be used to monitor mouse clicks. GetKeyState can retrieve the physical state of a button. A_TimeIdleMouse and A_TimeIdlePhysical can work correctly (ignoring keyboard input or artificial input, respectively). You can determine whether a script is using the hook via the KeyHistory function or menu item. You can determine which hotkeys are using the hook via the ListHotkeys function or menu item. Related InstallKeybdHook, #UseHook, Hotkey, KeyHistory, GetKeyState, KeyWait Examples Installs the mouse hook unconditionally. InstallMouseHook InStr - Syntax & Usage | AutoHotkey v2 InStr Searches for a given occurrence of a string, from the left or the right. FoundPos := InStr(Haystack, Needle , CaseSense, StartingPos, Occurrence) Parameters Haystack Type: String The string whose content is searched. Needle Type: String The string to search for. CaseSense Type: Integer or String One of the following values (defaults to 0 if omitted): "On" or 1 (True): The search is case sensitive. "Off" or 0 (False): The letters A-Z are considered identical to their lowercase counterparts. "Locale": The search is case insensitive according to the rules of the current user's locale. For example, most English and Western European locales treat not only the letters A-Z as identical to their lowercase counterparts, but also non-ASCII letters like Ä and Ü as identical to theirs. Locale is 1 to 8 times slower than Off depending on the nature of the strings being compared. StartingPos Type: Integer Omit StartingPos to search the entire string. Otherwise, specify the position at which to start the search, where 1 is the first character, 2 is the second character, and so on. Negative values count from the end of Haystack, so -1 is the last character, -2 is the second-last, and so on. If Occurrence is omitted, a negative StartingPos causes the search to be conducted from right to left. However, StartingPos has no effect on the direction of the search if Occurrence is specified. For a right-to-left search, StartingPos specifies the position of the last character of the first potential occurence of Needle. For example, InStr("abc", "bc",, 2, +1) will find a match but InStr("abc", "bc",, 2, -1) will not. If the absolute value of StartingPos is greater than the length of Haystack, 0 is returned. Occurrence Type: Integer If Occurrence is omitted, it defaults to the first match of Needle in Haystack. The search is conducted from right to left if StartingPos is negative; otherwise it is conducted from left to right. If Occurrence is positive, the search is always conducted from left to right. Specify 2 for Occurrence to return the position of the second match, 3 for the third match, etc. If Occurrence is negative, the search is always conducted from right to left. For example, -2 searches for the second occurrence from the right. Return Value Type: Integer This function returns the position of an occurrence of the string Needle in the string Haystack. Position 1 is the first character; this is because 0 is synonymous with "false", making it an intuitive "not found" indicator. Regardless of the values of StartingPos or Occurrence, the return value is always relative to the first character of Haystack. For example, the position of "abc" in "123abc789" is always 4. Conventionally, an occurrence of an empty string ("") can be found at any position. However, as a blank Needle would typically only by passed by mistake, it is treated as an error (an exception is thrown). Error Handling A ValueError is thrown in any of the following cases: Needle is an empty (zero-length) string. CaseSense is invalid. Occurrence or StartingPos is 0 or non-numeric. Remarks RegExMatch can be used to search for a pattern (regular expression) within a string, making it much more flexible than InStr. However, InStr is generally faster than RegExMatch when searching for a simple substring. InStr searches only up to the first binary zero (null-terminator), whereas RegExMatch searches the entire length of the string even if it includes binary zero. Related RegExMatch, Is functions Examples Reports the 1-based position of the substring "abc" in the string "123abc789". MsgBox InStr("123abc789", "abc") ; Returns 4 Searches for Needle in Haystack. Haystack := "The Quick Brown Fox Jumps Over the Lazy Dog" Needle := "Fox" If InStr(Haystack, Needle) MsgBox "The string was found." Else MsgBox "The string was not found." Demonstrates the difference between a case insensitive and case sensitive search. Haystack := "The Quick Brown Fox Jumps Over the Lazy Dog" Needle := "the" MsgBox InStr(Haystack, Needle, false, 1, 2) ; case insensitive search, return start position of second occurence MsgBox InStr(Haystack, Needle, true) ; case sensitive search, return start position of first occurence, same result as above Integer - Syntax & Usage | AutoHotkey v2 Integer Converts a numeric string or floating-point value to an integer. IntValue := Integer(Value) Return Value Type: Integer This function returns the result of converting Value to a pure integer (having the type name "Integer"), or Value itself if it is already the correct type. Remarks Any fractional part of Value is dropped, equivalent to Value < 0 ? Ceil(Value) : Floor(Value). If the value cannot be converted, a TypeError is thrown. To determine if a value can be converted to an integer, use the IsNumber function. Integer is actually a class, but can be called as a function. Value is Integer can be used to check whether a value is a pure integer. Related Type, Float, Number, String, Values, Expressions, Is functions List of Is Functions - Syntax & Usage | AutoHotkey v2 Is Functions Result := IsSomething(Value , Mode) These functions perform various checks. There are three categories: Type: Check the type of a value, or whether a string can be interpreted as a value of that type. Misc: Check miscellaneous conditions based on a given value or variable reference. String: Check that a string matches a specific pattern. Value must be a string, otherwise a TypeError is thrown. Mode is valid only for IsAlpha, IsAlnum, IsUpper and IsLower. By default, only ASCII letters are considered. To instead perform the check according to the rules of the current user's locale, specify the string "Locale" for the Mode parameter. The default mode can also be used by specifying 0 or 1. Type Check the type of a value, or whether a string can be interpreted as a value of that type. Function Description IsInteger True if Value is an integer or a purely numeric string (decimal or hexadecimal) without a decimal point. Leading and trailing spaces and tabs are allowed. The string may start with a plus or minus sign and must not be empty. IsFloat True if Value is a floating point number or a purely numeric string containing a decimal point. Leading and trailing spaces and tabs are allowed. The string may start with a plus sign, minus sign, or decimal point and must not be empty. IsNumber True if IsInteger(Value) or IsFloat(Value) is true. IsObject True if Value is an object. This includes objects derived from Object, prototype objects such as 0.base, and COM objects, but not numbers or strings. Misc Check miscellaneous conditions based on a given value or variable reference. Function Description IsLabel True if Value is the name of a label defined within the current scope. IsSet True if the variable Value has been assigned a value. IsSetRef True if the VarRef contained by Value has been assigned a value. String Check that a string matches a specific pattern. Value must be a string, otherwise a TypeError is thrown. Function Description IsDigit True if Value is a positive integer, an empty string, or a string which contains only the characters 0 through 9. Other

characters such as the following are not allowed: spaces, tabs, plus signs, minus signs, decimal points, hexadecimal digits, and the 0x prefix. IsXDigit Hexadecimal digit: Same as digit except the characters A through F (uppercase or lowercase) are also allowed. A prefix of 0x is tolerated if present. IsAlpha True if Value is a string and is empty or contains only alphabetic characters. False if there are any digits, spaces, tabs, punctuation, or other non-alphabetic characters anywhere in the string. For example, if Value contains a space followed by a letter, it is not considered to be alpha. By default, only ASCII letters are considered. To instead perform the check according to the rules of the current user's locale, use IsAlpha(Value, "Locale"). IsUpper True if Value is a string and is empty or contains only uppercase characters. False if there are any digits, spaces, tabs, punctuation, or other non-uppercase characters anywhere in the string. By default, only ASCII letters are considered. To instead perform the check according to the rules of the current user's locale, use IsUpper(Value, "Locale"). IsLower True if Value is a string and is empty or contains only lowercase characters. False if there are any digits, spaces, tabs, punctuation, or other non-lowercase characters anywhere in the string. By default, only ASCII letters are considered. To instead perform the check according to the rules of the current user's locale, use IsLower(Value, "Locale"). IsAlnum Same as IsAlpha except that integers and characters 0 through 9 are also allowed. IsSpace True if Value is a string and is empty or contains only whitespace consisting of the following characters: space (A_Space or `s), tab (A_Tab or `t), linefeed (`n), return (`r), vertical tab (`v), and formfeed (`f). IsTime True if Value is a valid date-time stamp, which can be all or just the leading part of the YYYYMMDDHH24MISS format. For example, a 4-digit string such as 2004 is considered valid. Use StrLen to determine whether additional time components are present. Value must have an even number of digits between 4 and 14 (inclusive) to be considered valid. Years less than 1601 are not considered valid because the operating system generally does not support them. The maximum year considered valid is 9999. Remarks Since literal numbers such as 128, 0x7F and 1.0 are converted to pure numbers before the script begins executing, the format of the literal number is lost. To avoid confusion, the string functions listed above throw an exception if they are given a pure number. Related A_YYYY, FileGetTime, If, StrLen, InStr, StrUpper, DateAdd Examples Checks whether var is a floating point number or an integer and checks whether it is a valid timestamp. if isFloat(var) MsgBox var " is a floating point number." else if isInteger(var) MsgBox var " is an integer." if isTime(var) MsgBox var " is also a valid date-time." IsLabel - Syntax & Usage | AutoHotkey v2 IsLabel Returns a non-zero number if the specified label exists in the current scope. IsLabel := IsLabel(LabelName) Parameters LabelName Type: String The name of a label. Return Value Type: Integer (boolean) This function returns 1 (true) if the specified label exists within the current scope, or 0 (false) if not. Remarks This function is useful to avoid runtime errors when specifying a dynamic label for Goto. When called from inside a function, only that function's labels are searched. Global labels are not valid targets for a local goto. Related Labels Examples Reports "Target label exists" because the label does exist. if IsLabel("Label") MsgBox "Target label exists" else MsgBox "Target label doesn't exist" Label: return IsObject - Syntax & Usage | AutoHotkey v2 IsObject Returns a non-zero number if the specified value is an object. TrueOrFalse := IsObject(Value) Parameters Value Type: Any The value to check. Return Value Type: Integer (boolean) This function returns 1 (true) if the specified value is an object, or 0 (false) if not. Remarks Any value which is not a primitive value (number or string) is considered to be an object, including those which do not derive from Object, such as COM wrapper objects. This distinction is made because objects share several common traits in contrast to primitive values: Each object is dynamically allocated and reference-counted. Any number of variables, properties or array elements may refer to the same object. For immutable values this distinction isn't important, but objects can have mutable properties. Each object has a unique address which is also an interface pointer compatible with IDispatch. An object compares equal to another value only if it is the same object. An object cannot be implicitly converted to a string or number. Related Objects Examples Reports "This is an object." because the value is an object. obj := {key: "value"} if IsObject(obj) MsgBox "This is an object." else MsgBox "This is not an object." IsSet / IsSetRef - Syntax & Usage | AutoHotkey v2 IsSet / IsSetRef Returns a non-zero number if the specified variable has been assigned a value. VarIsSet := IsSet(Var) VarIsSet := IsSetRef(&Ref) Parameters Var Type: Variable A direct variable reference. For example: IsSet(MyVar). &Ref Type: VarRef An indirect reference to the variable. This would usually not be passed directly, as in IsSetRef(&MyVar), but indirectly, such as to check a parameter containing a VarRef prior to dereferencing it. Return Value Type: Integer (boolean) The return value is 1 (true) if Var or the variable represented by Ref has been assigned a value, otherwise 0 (false). Remarks Use IsSet to check a variable directly, as in IsSet (MyGlobalVar). Use IsSetRef to check a VarRef, which would typically be contained by a variable, as in the example below. A variable which has not been assigned a value is also known as an uninitialized variable. Attempting to read an uninitialized variable causes an exception to be thrown. IsSet can be used to avoid this, such as for initializing a global or static variable on first use. Note: Static initializers such as static my_static_array := [] are evaluated only once, the first time they are reached during execution, so typically do not require the use of IsSet. Although IsSet uses the same syntax as a function call, it may be considered more of an operator than a function. The keyword IsSet is reserved for the use shown here and cannot be redefined as a variable or function. IsSet cannot be called indirectly because any attempt to pass an uninitialized variable would cause an error to be thrown. IsSetRef can also be used to check a specific variable, by using it with the reference operator. When using it this way, be aware of the need to declare the variable first if it is global. For example, the & in IsSetRef(&MyVar) would cause MyVar to resolve to a local variable by default, if used within an assume-local function which lacks the declaration global MyVar. Related ByRef parameters Examples Shows different uses for IsSet and IsSetRef. Loop 2 if !IsSet(MyVar) ; Is this the first "use" of MyVar? MyVar := A_Index ; Initialize on first "use". MsgBox Function1(&MyVar) MsgBox Function2(&MyVar) Function1(&Param) ; ByRef parameter. { if IsSet(Param) ; Pass Param itself, which is an alias for MyVar. return Param ; ByRef parameters are automatically dereferenced. else return "unset" } Function2(Param) { if IsSetRef(Param) ; Pass the VarRef contained by Param. return %Param% ; Explicitly dereference Param. else return "unset" } KeyHistory - Syntax & Usage | AutoHotkey v2 KeyHistory Displays script info and a history of the most recent keystrokes and mouse clicks. KeyHistory MaxEvents Parameters MaxEvents Type: Integer Omit this parameter to show the script's main window, equivalent to selecting the "View->Key history" menu item. Otherwise, this parameter sets the maximum number of keyboard and mouse events that can be recorded for display in the window (default 40, limit 500). The key history is also reset, but the main window is not shown or refreshed. Specify 0 to disable key history entirely. Remarks To disable key history, use the following: KeyHistory 0 This feature is intended to help debug scripts and hotkeys. It can also be used to detect the scan code of a non-standard keyboard key using the steps described at the bottom of the key list page (knowing the scan code allows such a key to be made into a hotkey). The virtual key (VK) of the wheel events (WheelDown, WheelUp, WheelLeft, and WheelRight) are placeholder values that do not have any meaning outside of AutoHotkey. Also, the scan code for wheel events is actually the number of notches by which the wheel was turned (typically 1). If the script does not have the keyboard hook installed, the KeyHistory window will display only the keyboard events generated by the script itself (i.e. not the user's). If the script does not have the mouse hook installed, mouse button events will not be shown. You can find out if your script uses either hook via "View->Key History" in the script's main window (accessible via "Open" in the tray icon). You can force the hooks to be installed by adding either or both of the following lines to the script: InstallKeybdHook InstallMouseHook Because each keystroke or mouse click consists of a down-event and an up-event, KeyHistory displays only half as many "complete events" as specified by MaxEvents. For example, if the script calls KeyHistory 50, up to 25 keystrokes and mouse clicks will be displayed. Related InstallKeybdHook, InstallMouseHook, ListHotkeys, ListLines, ListVars, GetKeyState, KeyWait, A_PriorKey Examples Displays the history info in a window. KeyHistory Causes KeyHistory to display the last 100 instead 40 keyboard and mouse events. KeyHistory 100 Disables key history entirely. KeyHistory 0 KeyWait - Syntax & Usage | AutoHotkey v2 KeyWait Waits for a key or mouse/joystick button to be released or pressed down. KeyWait KeyName , Options Parameters KeyName Type: String This can be just about any single character from the keyboard or one of the key names from the key list, such as a mouse/joystick button. Joystick attributes other than buttons are not supported. An explicit virtual key code such as vkFF may also be specified. This is useful in the rare case where a key has no name and produces no visible character when pressed. Its virtual key code can be determined by following the steps at the bottom of the key list page. Options Type: String If this parameter is blank, the function will wait indefinitely for the specified key or mouse/joystick button to be physically released by the user. However, if the keyboard hook is not installed and KeyName is a keyboard key released artificially by means such as the Send function, the key will be seen as having been physically released. The same is true for mouse buttons when the mouse hook is not installed. Options: A string of one or more of the following letters (in any order, with optional spaces in between): D: Wait for the key to be pushed down. L: Check the logical state of the key, which is the state that the OS and the active window believe the key to be in (not necessarily the same as the physical state). This option is ignored for joystick buttons. T: Timeout (e.g. T3). The number of seconds to wait before timing out and returning 0. If the key or button achieves the specified state, the function will not wait for the timeout to expire. Instead, it will immediately return 1. The timeout value can be a floating point number such as 2.5, but it should not be a hexadecimal value such as 0x03. Return Value Type: Integer (boolean) This function returns 0 (false) if the function timed out or 1 (true) otherwise. Remarks The physical state of a key or mouse button will usually be the same as the logical state unless the keyboard and/or mouse hooks are installed, in which case it will accurately reflect whether or not the user is physically holding down the key. You can determine if your script is using the hooks via the KeyHistory function or menu item. You can force either or both of the hooks to be installed by adding the InstallKeybdHook and InstallMouseHook functions to the script. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. To wait for two or more keys to be released, use KeyWait consecutively. For example: KeyWait "Control" ; Wait for both Control and Alt to be released. KeyWait "Alt" To wait for any one key among a set of keys to be pressed down, see the examples section of the InputHook function. Related GetKeyState, Key List, InputHook, KeyHistory, InstallKeybdHook, InstallMouseHook, ClipWait, WinWait Examples Waits for the A key to be released. KeyWait "a" Waits for the left mouse button to be pressed down. KeyWait "LButton", "D" Waits up to 3 seconds for the first joystick button to be pressed down. KeyWait "Joy1", "D T3" Waits for the left Alt key to be logically released. KeyWait "LAlt", "L" When pressing this hotkey, KeyWait waits for the user to physically release the CapsLock key. As a result, subsequent statements are performed on release instead of press. This behavior is similar to ~CapsLock up::. ~CapsLock:: { KeyWait "CapsLock" ; Wait for user to physically release it. MsgBox "You pressed and released the CapsLock key." } Remaps a key or mouse button (this is only for illustration because it would be easier to use the built-in remapping feature). The left mouse button is kept held down while NumpadAdd is down, which effectively transforms NumpadAdd into the left mouse button. *NumpadAdd:: { MouseClick "left",,, 1, 0, "D" ; Hold down the left mouse button.

KeyWait "NumpadAdd" ; Wait for the key to be released. MouseClick "left",,, 1, 0, "U" ; Release the mouse button. } Detects when a key has been double-pressed (similar to double-click). KeyWait is used to stop the keyboard's auto-repeat feature from creating an unwanted double-press when you hold down the RControl key to modify another key. It does this by keeping the hotkey's thread running, which blocks the auto-repeats by relying upon #MaxThreadsPerHotkey being at its default setting of 1. Note: There is a more elaborate script to distinguish between single, double, and triple-presses at the bottom of the SetTimer page. ~RControl:: { if (A_PriorHotkey != "~RControl" or A_TimeSincePriorHotkey > 400) { ; Too much time between presses, so this isn't a double-press. KeyWait "RControl" return } MsgBox "You double-pressed the right control key." } ListHotkeys - Syntax & Usage | AutoHotkey v2 ListHotkeys Displays the hotkeys in use by the current script, whether their subroutines are currently running, and whether or not they use the keyboard or mouse hook. ListHotkeys This function is equivalent to selecting the View->Hotkeys menu item in the main window. If a hotkey has been disabled via the Hotkey function, it will be listed as OFF or PART ("PART" means that only some of the hotkey's variants are disabled). If any of a hotkey's variants have a non-zero #InputLevel, the level (or minimum and maximum levels) are displayed. If any of a hotkey's subroutines are currently running, the total number of threads is displayed for that hotkey. Finally, the type of hotkey is also displayed, which is one of the following: reg: The hotkey is implemented via the operating system's RegisterHotkey() function. reg(no): Same as above except that this hotkey is inactive (due to being unsupported, disabled, or suspended). k-hook: The hotkey is implemented via the keyboard hook. m-hook: The hotkey is implemented via the mouse hook. 2-hooks: The hotkey requires both the hooks mentioned above. joypoll: The hotkey is implemented by polling the joystick at regular intervals. Related InstallKeybdHook, InstallMouseHook, #UseHook, KeyHistory, ListLines, ListVars, #MaxThreadsPerHotkey, A_MaxHotkeysPerInterval Examples Displays information about the hotkeys used by the current script. ListHotkeys ListLines - Syntax & Usage | AutoHotkey v2 ListLines Enables or disables line logging or displays the script lines most recently executed. ListLines Mode Parameters Mode Type: Integer If omitted, the history of lines most recently executed is shown. The first parameter affects only the behavior of the current thread as follows: 1 or True: Includes subsequently-executed lines in the history. This is the starting default for all scripts. 0 or False: Omits subsequently-executed lines from the history. Return Value Type: Integer This function returns the previous setting. Remarks ListLines (with no parameter) is equivalent to selecting the "View->Lines most recently executed" menu item in the main window. It can help debug a script. ListLines Mode can be used to selectively omit some lines from the history, which can help prevent the history from filling up too quickly (such as in a loop with many fast iterations). The line which called ListLines is also removed from the line history, to prevent clutter. Additionally, performance may be reduced by a few percent while line logging is enabled. When the ListLines mode is changed, the current line (generally the one that called ListLines or assigned to A_ListLines) is omitted from the line history. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. The built-in variable A_ListLines contains 1 if ListLines is enabled and 0 otherwise. On a related note, the built-in variables A_LineNumber and A_LineFile contain the currently executing line number and the file name to which it belongs. Related KeyHistory, ListHotkeys, ListVars Examples Enables and disables line logging for specific lines and then displays the result. x := "This line is logged" ListLines False x := "This line is not logged" ListLines True ListLines MsgBox ListVars - Syntax & Usage | AutoHotkey v2 ListVars Displays the script's variables: their names and current contents. ListVars Remarks This function is equivalent to selecting the "View->Variables" menu item in the main window. It can help you debug a script. For each variable in the list, the variable's name and contents are shown, along with other information depending on what the variable contains. Each item is terminated with a carriage return and newline (`r`n), but may span multiple lines if the variable contains `r`n. List items may take the following forms (where words in italics are placeholders): VarName[Length of Capacity]: String VarName: TypeName object {Info} VarName: Number Capacity is the variable's current capacity. String is the first 60 characters of the variable's string value. Info depends on the type of object, but is currently very limited. If ListVars is used inside a function, the following are listed: Local variables, including variables of outer functions which are referenced by the current function. Static variables for the current function. Global variables declared inside the function are also listed in this section. If the current function is nested inside another function, static variables of each outer function are also listed. All global variables. Related KeyHistory, ListHotkeys, ListLines The DebugVars script can be used to inspect and change the contents of variables and objects. Examples Displays information about the script's variables. var1 := "foo" var2 := "bar" obj := [] ListVars Pause ListView (GUI) - Syntax & Usage | AutoHotkey v2 ListView Table of Contents Introduction and Simple Example Options and Styles for the Options Parameter View Modes: Report (default), Icon, Tile, IconSmall, and List. Built-in Methods for ListViews Events ImageLists (the means by which icons are added to a ListView) Remarks Examples Introduction and Simple Example A List-View is one of the most elaborate controls provided by the operating system. In its most recognizable form, it displays a tabular view of rows and columns, the most common example of which is Explorer's list of files and folders (detail view). A ListView usually looks like this: Though it may be elaborate, a ListView's basic features are easy to use. The syntax for creating a ListView is: LV := GuiObj.Add ("ListView", Options, ["ColumnTitle1","ColumnTitle2","..."]) Or: LV := GuiObj.AddListView(Options, ["ColumnTitle1","ColumnTitle2","..."]) Here is a working script that creates and displays a ListView containing a list of files in the user's "My Documents" folder: ; Create the window: MyGui := Gui() ; Create the ListView with two columns, Name and Size: LV := MyGui.Add("ListView", "r20 w700", ["Name","Size (KB)"]) ; Notify the script whenever the user double clicks a row: LV.OnEvent("DoubleClick", LV_DoubleClick) ; Gather a list of file names from a folder and put them into the ListView: Loop Files, A_MyDocuments "*.*" LV.Add(, A_LoopFileName, A_LoopFileSizeKB) LV.ModifyCol ; Auto-size each column to fit its contents. LV.ModifyCol(2, "Integer") ; For sorting purposes, indicate that column 2 is an integer. ; Display the window: MyGui.Show LV_DoubleClick(LV, RowNumber) { RowText := LV.GetText(RowNumber) ; Get the text from the row's first field. ToolTip("You double-clicked row number " RowNumber ". Text: '" RowText "'") } Options and Styles for the Options Parameter Background: Specify the word Background followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: BackgroundSilver, BackgroundFFDD99. If this option is not present, the ListView initially defaults to the system's default background color. Specifying BackgroundDefault or -Background applies the system's default background color (usually white). For example, a ListView can be restored to the default color via LV.Opt("+BackgroundDefault"). C: Text color. Specify the letter C followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: cRed, cFF2211, c0xFF2211, cDefault. Checked: Provides a checkbox at the left side of each row. When adding a row, specify the word Check in its options to have the box to start off checked instead of unchecked. The user may either click the checkbox or press the spacebar to check or uncheck a row. Count: Specify the word Count followed immediately by the total number of rows that the ListView will ultimately contain. This is not a limit: rows beyond the count can still be added. Instead, this option serves as a hint to the control that allows it to allocate memory only once rather than each time a row is added, which greatly improves row-adding performance (it may also improve sorting performance). To improve performance even more, use LV.Opt("-Redraw") prior to adding a large number of rows and LV.Opt("+Redraw") afterward. See Redraw for more details. Grid: Provides horizontal and vertical lines to visually indicate the boundaries between rows and columns. Hdr: Specify -Hdr (minus Hdr) to omit the header (the special top row that contains column titles). To make it visible later, use LV.Opt ("+Hdr"). LV: Specify the string LV followed immediately by the number of an extended ListView style. These styles are entirely separate from generic extended styles. For example, specifying -E0x200 would remove the generic extended style WS_EX_CLIENTEDGE to eliminate the control's default border. By contrast, specifying -LV0x20 would remove LVS_EX_FULLROWSELECT. LV0x10: Specify -LV0x10 to prevent the user from dragging column headers to the left or right to reorder them. However, it is usually not necessary to do this because the physical reordering of columns does not affect the column order seen by the script. For example, the first column will always be column 1 from the script's point of view, even if the user has physically moved it to the right of other columns. LV0x20: Specify -LV0x20 to require that a row be clicked at its first field to select it (normally, a click on any field will select it). The advantage of this is that it makes it easier for the user to drag a rectangle around a group of rows to select them. Multi: Specify -Multi (minus Multi) to prevent the user from selecting more than one row at a time. NoSortHdr: Prevents the header from being clickable. It will take on a flat appearance rather than its normal button-like appearance. Unlike most other ListView styles, this one cannot be changed after the ListView is created. NoSort: Turns off the automatic sorting that occurs when the user clicks a column header. However, the header will still behave visually like a button (unless NoSortHdr has been specified). In addition, the ColClick event is still raised, so the script can respond with a custom sort or other action. ReadOnly: Specify -ReadOnly (minus ReadOnly) to allow editing of the text in the first column of each row. To edit a row, select it then press F2 (see the WantF2 option below). Alternatively, you can click a row once to select it, wait at least half a second, then click the same row again to edit it. R: Rows of height (upon creation). Specify the letter R followed immediately by the number of rows for which to make room inside the control. For example, R10 would make the control 10 rows tall. If the ListView is created with a view mode other than report view, the control is sized to fit rows of icons instead of rows of text. Note: adding icons to a ListView's rows will increase the height of each row, which will make this option inaccurate. Sort: The control is kept alphabetically sorted according to the contents of the first column. SortDesc: Same as above except in descending order. WantF2: Specify -WantF2 (minus WantF2) to prevent F2 from editing the currently focused row. This setting is ignored unless -ReadOnly is also in effect. (Unnamed numeric styles): Since styles other than the above are rarely used, they do not have names. See the ListView styles table for a list. View Modes A ListView has five viewing modes, of which the most common is report view (which is the default). To use one of the other views, specify its name in the options list. The view can also be changed after the control is created; for example: LV.Opt("+IconSmall"). Icon: Shows a large-icon view. In this view and all the others except Report, the text in columns other than the first is not visible. To display icons in this mode, the ListView must have a large-icon ImageList assigned to it. Tile: Shows a large-icon view but with ergonomic differences such as displaying each item's text to the right of the icon rather than underneath it. Checkboxes do not function in this view. IconSmall: Shows a small-icon view. List: Shows a small-icon view in list format, which displays the icons in columns. The number of columns depends on the width of the control and the width of the widest text item in it. Report: Switches back to report view, which is the initial default. For example: LV.Opt("+Report"). Built-in Methods for ListViews In addition to the default methods/properties of a GUI control, ListView controls have the following methods (defined in the Gui.ListView class). When the phrase "row number" is used on this page, it refers to a row's current position within the ListView. The top row is 1, the second row is 2, and so on. After a row is added,

its row number tends to change due to sorting, deleting, and inserting of other rows. Therefore, to locate specific row(s) based on their contents, it is usually best to use the GetText method in a loop. Row methods: Add: Adds a new row to the bottom of the list. Insert: Inserts a new row at the specified row number. Modify: Modifies the attributes and/or text of a row. Delete: Deletes the specified row or all rows. Column methods: ModifyCol: Modifies the attributes and/or text of the specified column and its header. InsertCol: Inserts a new column at the specified column number. DeleteCol: Deletes the specified column and all of the contents beneath it. Retrieval methods: GetCount: Returns the total number of rows or columns. GetNext: Returns the row number of the next selected, checked, or focused row. GetText: Retrieves the text at the specified row and column. Other methods: SetImageList: Sets or replaces an ImageList for displaying icons. Add Adds a new row to the bottom of the list, and returns the new row number, which is not necessarily the last row if the ListView has the Sort or SortDesc style. NewRowNumber := LV.Add(Options, Col1, Col2, ...) Options Type: String A string containing zero or more words from the list below (not case sensitive). Separate each word from the next with a space or tab. To remove an option, precede it with a minus sign. To add an option, a plus sign is permitted but not required. Check: Shows a checkmark in the row (if the ListView has checkboxes). To later uncheck it, use LV.Modify(RowNumber, "-Check"). Col: Specify the word Col followed immediately by the column number at which to begin applying the parameters Col1 and beyond. This is most commonly used with the Modify method to alter individual fields in a row without affecting those that lie to their left. Focus: Sets keyboard focus to the row (often used in conjunction with Select). To later de-focus it, use LV.Modify(RowNumber, "-Focus"). Icon: Specify the word Icon followed immediately by the number of this row's icon, which is displayed in the left side of the first column. If this option is absent, the first icon in the ImageList is used. To display a blank icon, specify -1 or a number that is larger than the number of icons in the ImageList. If the control lacks a small-icon ImageList, no icon is displayed nor is any space reserved for one in report view. This option accepts a one-based icon number, but this is internally translated to a zero-based index; therefore, Icon0 corresponds to the constant I_IMAGECALLBACK, which is normally defined as -1, and Icon-1 corresponds to I_IMAGENONE. Other out of range values may also cause a blank space where the icon would be. Select: Selects the row. To later deselect it, use LV.Modify(RowNumber, "-Select"). When selecting rows, it is usually best to ensure that at least one row always has the focus property because that allows the Apps key to display its context menu (if any) near the focused row. The word Select may optionally be followed immediately by a 0 or 1 to indicate the starting state. In other words, both "Select" and "Select" . VarContainingOne are the same (the period used here is the concatenation operator). This technique also works with Focus and Check above. Vis: Ensures that the specified row is completely visible by scrolling the ListView, if necessary. This has an effect only for LV.Modify; for example: LV.Modify(RowNumber, "Vis"). Col1, Col2, ... Type: String The columns of the new row, which can be text or numeric (including numeric expression results). To make any field blank, specify "" or the equivalent. If there are too few fields to fill all the columns, the columns at the end are left blank. If there are too many fields, the fields at the end are completely ignored. Insert Inserts a new row at the specified row number, and returns the new row number. NewRowNumber := LV.Insert(RowNumber , Options, Col1, Col2, ...) RowNumber Type: Integer The row number for the newly inserted row. Any rows at or beneath RowNumber are shifted downward to make room for the new row. If RowNumber is greater than the number of rows in the list (even as high as 2147483647), the new row is added to the end of the list. Options Type: String See row options. Col1, Col2, ... Type: String The columns of the new row, which can be text or numeric (including numeric expression results). To make any field blank, specify "" or the equivalent. If there are too few fields to fill all the columns, the columns at the end are left blank. If there are too many fields, the fields at the end are completely ignored. Modify Modifies the attributes and/or text of a row. LV.Modify(RowNumber , Options, NewCol1, NewCol2, ...) Note: When only the first two parameters are present, only the row's attributes and not its text are changed. RowNumber Type: Integer The row to modify. If RowNumber is 0, all rows in the control are modified (in this case the return value is 1 on complete success and 0 if any part of the operation failed). Options Type: String The ColN option may be used to update specific columns without affecting the others. For other options, see row options. NewCol1, NewCol2, ... Type: String The new columns of the specified row, which can be text or numeric (including numeric expression results). To make any field blank, specify "" or the equivalent. If there are too few parameters to cover all the columns, the columns at the end are not changed. If there are too many fields, the fields at the end are completely ignored. Delete Deletes the specified row or all rows. LV.Delete(RowNumber) RowNumber Type: Integer The row to delete. If this parameter is omitted, all rows in the ListView are deleted. ModifyCol Modifies the attributes and/or text of the specified column and its header. LV.ModifyCol(ColumnNumber, Options, ColumnTitle) Note: If all parameters are omitted, the width of every column is adjusted to fit the contents of the rows. If only the first parameter is present, only the specified column is auto-sized. Auto-sizing has no effect when not in Report (Details) view. ColumnNumber Type: Integer The column to modify. The first column is number 1 (not 0). Options Type: String A string containing zero or more words from the list below (not case sensitive). Separate each word from the next with a space or tab. To remove an option, precede it with a minus sign. To add an option, a plus sign is permitted but not required. General options: N: Specify for N the new width of the column, in pixels. This number can be unquoted if is the only option. For example, the following are both valid: LV.ModifyCol(1, 50) and LV.ModifyCol(1, "50 Integer"). Auto: Adjusts the column's width to fit its contents. This has no effect when not in Report (Details) view. AutoHdr: Adjusts the column's width to fit its contents and the column's header text, whichever is wider. If applied to the last column, it will be made at least as wide as all the remaining space in the ListView. It is usually best to apply this setting only after the rows have been added because that allows any newly-arrived vertical scroll bar to be taken into account when sizing the last column. This option has no effect when not in Report (Details) view. Icon: Specify the word Icon followed immediately by the number of the ImageList's icon to display next to the column header's text. Specify -Icon (minus icon) to remove any existing icon. IconRight: Puts the icon on the right side of the column rather than the left. Data type options: Float: For sorting purposes, indicates that this column contains floating point numbers (hexadecimal format is not supported). Sorting performance for Float and Text columns is up to 25 times slower than it is for integers. Integer: For sorting purposes, indicates that this column contains integers. To be sorted properly, each integer must be 32-bit; that is, within the range -2147483648 to 2147483647. If any of the values are not integers, they will be considered zero when sorting (unless they start with a number, in which case that number is used). Numbers may appear in either decimal or hexadecimal format (e.g. 0xF9E0). Text: Changes the column back to text-mode sorting, which is the initial default for every column. Only the first 8190 characters of text are significant for sorting purposes (except for the Logical option, in which case the limit is 4094). Alignment options: Center: Centers the text in the column. To center an Integer or Float column, specify the word Center after the word Integer or Float. Left: Left-aligns the column's text, which is the initial default for every column. On older operating systems, the first column might have a forced left-alignment. Right: Right-aligns the column's text. This attribute need not be specified for Integer and Float columns because they are right-aligned by default. That default can be overridden by specifying something such as "Integer Left" or "Float Center". Sorting options: Case: The sorting of the column is case sensitive (affects only text columns). If the options Case, CaseLocale, and Logical are all omitted, the uppercase letters A-Z are considered identical to their lowercase counterparts for the purpose of the sort. CaseLocale: The sorting of the column is case insensitive based on the current user's locale (affects only text columns). For example, most English and Western European locales treat the letters A-Z and ANSI letters like Ä and Ü as identical to their lowercase counterparts. This method also uses a "word sort", which treats hyphens and apostrophes in such a way that words like "coop" and "co-op" stay together. Desc: Descending order. The column starts off in descending order the first time the user sorts it. Logical: Same as CaseLocale except that any sequences of digits in the text are treated as true numbers rather than mere characters. For example, the string "T33" would be considered greater than "T4". Logical and Case are currently mutually exclusive: only the one most recently specified will be in effect. NoSort: Prevents a user's click on this column from having any automatic sorting effect. However, the ColClick event is still raised, so the script can respond with a custom sort or other action. To disable sorting for all columns rather than only a subset, include NoSort in the ListView's options. Sort: Immediately sorts the column in ascending order (even if it has the Desc option). SortDesc: Immediately sorts the column in descending order. Uni: Unidirectional sort. This prevents a second click on the same column from reversing the sort direction. ColumnTitle Type: String The new column header. Omit this parameter to left it unchanged. InsertCol Inserts a new column at the specified column number, and returns the new column's position number. NewColumnNumber := LV.InsertCol(ColumnNumber , Options, ColumnTitle) ColumnNumber Type: Integer The column number of the newly inserted column. Any column at or on the right side of ColumnNumber are shifted to the right to make room for the new column. The first column is 1 (not 0). The maximum number of columns in a ListView is 200. If ColumnNumber is larger than the number of columns currently in the control, the new column is added next to the last column on the right side. The newly inserted column starts off with empty contents beneath it unless it is the first column, in which case it inherits the old first column's contents and the old first column acquires blank contents. Options Type: String The new column's attributes -- such as whether or not it uses integer sorting -- always start off at their defaults unless changed via Options. ColumnTitle Type: String The new column header. DeleteCol Deletes the specified column and all of the contents beneath it. LV.DeleteCol(ColumnNumber) ColumnNumber Type: Integer The column to delete. Once a column is deleted, the column numbers of any that lie to its right are reduced by 1. Consequently, calling LV.DeleteCol(2) twice would delete the second and third columns. GetCount Returns the number of rows or columns in the control. CountNumber := LV.GetCount(Mode) Mode Type: String When this parameter is omitted, it returns the total number of rows in the control. When this parameter is "S" or "Selected", the count includes only the selected/highlighted rows. When this parameter is "Col" or "Column", it returns the number of columns in the control. The return value will be retrieved instantly, because the control keeps track of these counts. This method is often used in the top line of a Loop, in which case the method would get called only once (prior to the first iteration). For example: Loop LV.GetCount { RetrievedText := LV.GetText(A_Index) if InStr(RetrievedText, "some filter text") LV.Modify(A_Index, "Select") ; Select each row whose first field contains the filter-text. } To retrieve the widths of a ListView's columns -- for uses such as saving them to an INI file to be remembered between sessions -- follow this example: Loop LV.GetCount("Column") { ColWidth := SendMessage(0x101D, A_Index - 1, 0, LV) ; 0x101D is LVM_GETCOLUMNWIDTH. MsgBox("Column " A_Index "'s width is " ColWidth ".") } GetNext Returns the row number of the next selected, checked, or focused row, otherwise zero. RowNumber := LV.GetNext (StartingRowNumber, RowType) StartingRowNumber Type: Integer If omitted or less than 1, the search begins at the top of the list. Otherwise, the search begins at the row after StartingRowNumber. RowType Type: String If omitted, it searches for the next selected/highlighted row. Otherwise, specify "C" or "Checked" to

find the next checked row; or "F" or "Focused" to find the focused row (there is never more than one focused row in the entire list, and sometimes there is none at all). The following example reports all selected rows in the ListView: RowNumber := 0 ; This causes the first loop iteration to start the search at the top of the list. Loop { RowNumber := LV.GetNext(RowNumber) ; Resume the search at the row after that found by the previous iteration. if not RowNumber ; The above returned zero, so there are no more selected rows. break Text := LV.GetText(RowNumber) MsgBox('The next selected row is #' RowNumber ', whose first field is "' Text '".') } An alternate method to find out if a particular row number is checked is the following: ItemState := SendMessage(0x102C, RowNumber - 1, 0xF000, LV) ; 0x102C is LVM_GETITEMSTATE. 0xF000 is LVIS_STATEIMAGEMASK. IsChecked := (ItemState >> 12) - 1 ; This sets IsChecked to true if RowNumber is checked or false otherwise. GetText Retrieves the text at the specified row and column number. RetrievedText := LV.GetText(RowNumber , ColumnNumber) RowNumber Type: Integer The number of the row, whose text to be retrieved. If this parameter is 0, the column header text is retrieved. RetrievedText has a maximum length of 8191. ColumnNumber Type: Integer The number of the column, where the specified row is located. If omitted, it defaults to 1 (the text in the first column). Column numbers seen by the script are not altered by any dragging and dropping of columns the user may have done. For example, the original first column is still number 1 even if the user drags it to the right of other columns. SetImageList Sets or replaces an ImageList for displaying icons, and returns the ImageListID that was previously associated with this control (or 0 if none). PrevImageListID := LV.SetImageList(ImageListID , IconType) ImageListID Type: Integer The number returned from a previous call to IL_Create. IconType Type: Integer If the second parameter is omitted, the type of icons in the ImageList is detected automatically as large or small. Otherwise, specify 0 for large icons, 1 for small icons, and 2 for state icons (state icons are not yet directly supported, but they could be used via SendMessage). This method is normally called prior to adding any rows to the ListView. A ListView may have up to two ImageLists: small-icon and/or large-icon. This is useful when the script allows the user to switch to and from the large-icon view. To add more than one ImageList to a ListView, call the SetImageList method a second time, specifying the ImageListID of the second list. A ListView with both a large-icon and small-icon ImageList should ensure that both lists contain the icons in the same order. This is because the same ID number is used to reference both the large and small versions of a particular icon. Although it is traditional for all viewing modes except Icon and Tile to show small icons, this can be overridden by passing a large-icon list to the SetImageList method and specifying 1 (small-icon) for the second parameter. This also increases the height of each row in the ListView to fit the large icon. Any such detached ImageList should normally be destroyed via IL_Destroy. Events The following events can be detected by calling OnEvent to register a callback function or method: EventRaised when... ClickThe control is clicked. DoubleClickThe control is double-clicked. ColClickA column header is clicked. ContextMenuThe user right-clicks the control or presses Menu or Shift+F10 while the control has the keyboard focus. FocusThe control gains the keyboard focus. LoseFocusThe control loses the keyboard focus. ItemCheckAn item is checked or unchecked. ItemEditAn item's label is edited by the user. ItemFocusThe focused item changes. ItemSelectAn item is selected or deselected. Additional (rarely-used) notifications can be detected by using OnNotify. These notifications are documented at Microsoft Docs. Microsoft Docs does not show the numeric value of each notification code; those can be found in the Windows SDK or by searching the Internet. ImageLists An Image-List is a group of identically sized icons stored in memory. Upon creation, each ImageList is empty. The script calls IL_Add repeatedly to add icons to the list, and each icon is assigned a sequential number starting at 1. This is the number to which the script refers to display a particular icon in a row or column header. Here is a working example that demonstrates how to put icons into a ListView's rows: MyGui := Gui() ; Create a MyGui window. LV := MyGui.Add("ListView", "h200 w180", ["Icon & Number","Description"]) ; Create a ListView. ImageListID := IL_Create(10) ; Create an ImageList to hold 10 small icons. LV.SetImageList (ImageListID) ; Assign the above ImageList to the current ListView. Loop 10 ; Load the ImageList with a series of icons from the DLL. IL_Add(ImageListID, "shell32.dll", A_Index) Loop 10 ; Add rows to the ListView (for demonstration purposes, one for each icon). LV.Add("Icon" . A_Index, A_Index, "n/a") LV.ModifyCol("Hdr") ; Auto-adjust the column widths. MyGui.Show IL_Create Creates a new ImageList, initially empty, and returns the unique ID of the ImageList (or 0 upon failure). ImageListID := IL_Create(InitialCount, GrowCount, LargeIcons) InitialCount Type: Integer The number of icons you expect to put into the list immediately (if omitted, it defaults to 2). GrowCount Type: Integer The number of icons by which the list will grow each time it exceeds the current list capacity (if omitted, it defaults to 5). LargeIcons Type: Boolean If this parameter is 1 (true), the ImageList will contain large icons. If it is 0 (false), it will contain small icons (this is the default when omitted). Icons added to the list are scaled automatically to conform to the system's dimensions for small and large icons. IL_Add Adds an icon or picture to the specified ImageList, and returns the new icon's index (1 is the first icon, 2 is the second, and so on). IconIndex := IL_Add (ImageListID, Filename , IconNumber, ResizeNonIcon) ImageListID Type: Integer The ID of a ImageList created by IL_Create. Filename Type: String The name of an icon (.ICO), cursor (.CUR), animated cursor (.ANI) file (animated cursors will not actually be animated when displayed in a ListView), or a bitmap or icon handle like "HBITMAP:" handle. Other sources of icons include the following types of files: EXE, DLL, CPL, SCR, and other types that contain icon resources. IconNumber Type: Integer To use an icon group other than the first one in the file, specify its number for IconNumber. If IconNumber is negative, its absolute value is assumed to be the resource ID of an icon within an executable file. In the following example, the default icon from the second icon group would be used: IL_Add(ImageListID, "C:\My Application.exe", 2). ResizeNonIcon Type: Boolean Non-icon images such as BMP, GIF and JPG may also be loaded. However, in this case the last two parameters should be specified to ensure correct behavior: IconNumber should be the mask/transparency color number (0xFFFFFF [the color white] might be best for most pictures); and ResizeNonIcon should be non-zero to cause the picture to be scaled to become a single icon, or zero to divide up the image into however many icons can fit into its actual width. Supported image formats include ANI, BMP, CUR, EMF, Exif, GIF, ICO, JPG, PNG, TIF, and WMF. IL_Destroy Deletes the specified ImageList. Success := IL_Destroy(ImageListID) ImageListID Type: Integer The ID of a ImageList created by IL_Create. Note: It is normally not necessary to destroy ImageLists because once attached to a ListView, they are destroyed automatically when the ListView or its parent window is destroyed. However, if the ListView shares ImageLists with other ListViews (by having 0x40 in its options), the script should explicitly destroy the ImageList after destroying all the ListViews that use it. Similarly, if the script replaces one of a ListView's old ImageLists with a new one, it should explicitly destroy the old one. Remarks Gui.Submit has no effect on a ListView control. After a column is sorted -- either by means of the user clicking its header or the script calling LV.ModifyCol(1, "Sort") -- any subsequently added rows will appear at the bottom of the list rather than obeying the sort order. The exception to this is the Sort and SortDesc styles, which move newly added rows into the correct positions. To detect when the user has pressed Enter while a ListView has focus, use a default button (which can be hidden if desired). For example: MyGui.Add("Button", "Hidden Default", "OK").OnEvent("Click", LV_Enter) ... LV_Enter(*) { global if MyGui.FocusedCtrl != LV return MsgBox("Enter was pressed. The focused row number is " LV.GetNext(0, "Focused")) } In addition to navigating from row to row with the keyboard, the user may also perform incremental search by typing the first few characters of an item in the first column. This causes the selection to jump to the nearest matching row. Although any length of text can be stored in each field of a ListView, only the first 260 characters are displayed. Although the maximum number of rows in a ListView is limited only by available system memory, row-adding performance can be greatly improved as described in the Count option. A picture may be used as a background around a ListView (that is, to frame the ListView). To do this, create the picture control after the ListView and include 0x4000000 (which is WS_CLIPSIBLINGS) in the picture's Options. A script may create more than one ListView per window. It is best not to insert or delete columns directly with SendMessage. This is because the program maintains a collection of sorting preferences for each column, which would then get out of sync. Instead, use the built-in column methods. To perform actions such as resizing, hiding, or changing the font of a ListView, see GuiControl object. To extract text from external ListViews (those not owned by the script), use ListViewGetContent. Related TreeView, Other Control Types, Gui(), ContextMenu event, Gui object, GuiControl object, ListView styles table Examples Selects or de-selects all rows by specifying 0 as the row number. ; Select or de-select all rows by specifying 0 as the row number: LV.Modify(0, "Select") ; Select all. LV.Modify(0, "-Select") ; De-select all. LV.Modify(0, "-Check") ; Uncheck all the checkboxes. Auto-sizes all columns to fit their contents. LV.ModifyCol ; There are no parameters in this mode. The following is a working script that is more elaborate than the one near the top of this page. It displays the files in a folder chosen by the user, with each file assigned the icon associated with its type. The user can double-click a file, or right-click one or more files to display a context menu. ; Create a GUI window: MyGui := Gui("+Resize") ; Allow the user to maximize or drag-resize the window. ; Create some buttons: B1 := MyGui.Add("Button", "Default", "Load a folder") B2 := MyGui.Add("Button", "x+20", "Clear List") B3 := MyGui.Add("Button", "x+20", "Switch View") ; Create the ListView and its columns via MyGui.Add: LV := MyGui.Add("ListView", "xm r20 w700", ["Name","In Folder","Size (KB)","Type"]) LV.ModifyCol(3, "Integer") ; For sorting, indicate that the Size column is an integer. ; Create an ImageList so that the ListView can display some icons: ImageListID1 := IL_Create(10) ImageListID2 := IL_Create(10, 10, true) ; A list of large icons to go with the small ones. ; Attach the ImageLists to the ListView so that it can later display the icons: LV.SetImageList(ImageListID1) LV.SetImageList(ImageListID2) ; Apply control events: LV.OnEvent ("DoubleClick", RunFile) LV.OnEvent("ContextMenu", ShowContextMenu) B1.OnEvent("Click", LoadFolder) B2.OnEvent("Click", (*) => LV.Delete()) B3.OnEvent("Click", SwitchView) ; Apply window events: MyGui.OnEvent("Size", Gui_Size) ; Display the window: MyGui.Show() LoadFolder(*) { static IconMap := Map() MyGui.Opt("+OwnDialogs") ; Forces user to dismiss the following dialog before using main window. Folder := DirSelect(, 3, "Select a folder to read:") if not Folder ; The user canceled the dialog. return ; Check if the last character of the folder name is a backslash, which happens for root ; directories such as C:\. If it is, remove it to prevent a double-backslash later on. if SubStr(Folder, -1, 1) = "\" Folder := SubStr(Folder, 1, -1) ; Remove the trailing backslash. ; Calculate buffer size required for SHFILEINFO structure. sfi_size := A_PtrSize + 688 sfi := Buffer(sfi_size) ; Gather a list of file names from the selected folder and append them to the ListView: LV.Opt("-Redraw") ; Improve performance by disabling redrawing during load. Loop Files, Folder "*.*" { FileName := A_LoopFilePath ; Must save it to a writable variable for use below. ; Build a unique extension ID to avoid characters that are illegal in variable names, ; such as dashes. This unique ID method also performs better because finding an item ; in the array does not require search-loop. SplitPath(FileName,,, &FileExt) ; Get the file's extension. if FileExt ~= "^(EXE|ICO|ANI|CUR)$" { ExtID := FileExt ; Special ID as a placeholder. IconNumber := 0 ; Flag it as not found so that these types can each have a unique icon. } else ; Some other extension/file-type, so calculate its unique ID. { ExtID := 0 ; Initialize to handle extensions that are shorter than

others. Loop 7 ; Limit the extension to 7 characters so that it fits in a 64-bit value. { ExtChar := SubStr(FileExt, A_Index, 1) if not ExtChar ; No more characters. break ; Derive a Unique ID by assigning a different bit position to each character: ExtID := ExtID | (Ord(ExtChar) << (8 * (A_Index - 1))) } ; Check if this file extension already has an icon in the ImageLists. If it does, ; several calls can be avoided and loading performance is greatly improved, ; especially for a folder containing hundreds of files: IconNumber := IconMap.Has(ExtID) ? IconMap[ExtID] : 0 } if not IconNumber ; There is not yet any icon for this extension, so load it. { ; Get the high-quality small-icon associated with this file extension: if not DllCall("Shell32\SHGetFileInfoW", "Str", FileName , "Uint", 0, "Ptr", sfi, "UInt", sfi_size, "UInt", 0x101) ; 0x101 is SHGFI_ICON+SHGFI_SMALLICON IconNumber := 9999999 ; Set it out of bounds to display a blank icon. else ; Icon successfully loaded. { ; Extract the hIcon member from the structure: hIcon := NumGet(sfi, 0, "Ptr") ; Add the HICON directly to the small-icon and large-icon lists. ; Below uses +1 to convert the returned index from zero-based to one-based: IconNumber := DllCall("ImageList_ReplaceIcon", "Ptr", ImageListID1, "Int", -1, "Ptr", hIcon) + 1 DllCall("ImageList_ReplaceIcon", "Ptr", ImageListID2, "Int", -1, "Ptr", hIcon) ; Now that it's been copied into the ImageLists, the original should be destroyed: DllCall("DestroyIcon", "Ptr", hIcon) ; Cache the icon to save memory and improve loading performance: IconMap[ExtID] := IconNumber } } ; Create the new row in the ListView and assign it the icon number determined above: LV.Add("Icon" . IconNumber, A_LoopFileName, A_LoopFileDir, A_LoopFileSizeKB, FileExt) } LV.Opt("+Redraw") ; Re-enable redrawing (it was disabled above). LV.ModifyCol() ; Auto-size each column to fit its contents. LV.ModifyCol(3, 60) ; Make the Size column at little wider to reveal its header. } SwitchView(*) { static IconView := false if not IconView LV.Opt ("+Icon") ; Switch to icon view. else LV.Opt("+Report") ; Switch back to details view. IconView := not IconView ; Invert in preparation for next time. } RunFile (LV, RowNumber) { FileName := LV.GetText(RowNumber, 1) ; Get the text of the first field. FileDir := LV.GetText(RowNumber, 2) ; Get the text of the second field. try Run(FileDir "\" FileName) catch MsgBox("Could not open " FileDir "\" FileName ".") } ShowContextMenu(LV, Item, IsRightClick, X, Y) ; In response to right-click or Apps key. { ; Create the popup menu to be used as the context menu: ContextMenu := Menu() ContextMenu.Add("Open", OpenOrProperties) ContextMenu.Add("Properties", OpenOrProperties) ContextMenu.Add("Clear from ListView", ClearRows) ContextMenu.Default := "Open" ; Make "Open" a bold font to indicate that double-click does the same thing. ; Show the menu at the provided coordinates, X and Y. These should be used ; because they provide correct coordinates even if the user pressed the Apps key: ContextMenu.Show(X, Y) OpenOrProperties(ItemName, *) ; The user selected "Open" or "Properties" in the context menu. { FileName := LV.GetText(Item, 1) ; Get the text of the first field. FileDir := LV.GetText(Item, 2) ; Get the text of the second field. try { if (ItemName = "Open") ; User selected "Open" from the context menu. Run(FileDir "\" FileName) else Run("properties " FileDir "\" FileName) } catch MsgBox ("Could not perform requested action on " FileDir "\" FileName ".") } ClearRows(*) ; The user selected "Clear" in the context menu. { RowNumber := 0 ; This causes the first iteration to start the search at the top. Loop { ; Since deleting a row reduces the RowNumber of all other rows beneath it, ; subtract 1 so that the search includes the same row number that was previously ; found (in case adjacent rows are selected): RowNumber := LV.GetNext(RowNumber - 1) if not RowNumber ; The above returned zero, so there are no more selected rows. break LV.Delete(RowNumber) ; Clear the row from the ListView. } } } Gui_Size (thisGui, MinMax, Width, Height) ; Expand/Shrink ListView in response to the user's resizing. { if MinMax = -1 ; The window has been minimized. No action needed. return ; Otherwise, the window has been resized or maximized. Resize the ListView to match. LV.Move(„ Width - 20, Height - 40) } ListViewGetContent - Syntax & Usage | AutoHotkey v2 ListViewGetContent Returns a list of items/rows from a ListView. List := ListViewGetContent(Options, Control, WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Options Type: String Specifices what to retrieve. If blank or omitted, all the text in the ListView is retrieved. Otherwise, specify zero or more of the following words, each separated from the next with a space or tab: Selected: Returns only the selected (highlighted) rows rather than all rows. If none, the return value is blank. Focused: Returns only the focused row. If none, the return value is blank. Col4: Returns only the fourth column (field) rather than all columns (replace 4 with a number of your choice). Count: Returns a single number that is the total number of rows in the ListView. Count Selected: Returns the number of selected (highlighted) rows. Count Focused: Returns the row number (position) of the focused row (0 if none). Count Col: Returns the number of columns in the control (or -1 if the count cannot be determined). Control Type: String, Integer or Object The control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns a list of items/rows. Within each row, each field (column) except the last will end with a tab character (`t). To access the items/rows individually, use a parsing loop as in example #1. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if a message could not be sent to the control, or if the process owning the ListView could not be opened, perhaps due to a lack of user permissions or because it is locked. A ValueError is thrown if the ColN option specifies a nonexistent column. Remarks Some applications store their ListView text privately, which prevents their text from being retrieved. In these cases, an exception will usually not be thrown, but all the retrieved fields will be empty. The columns in a ListView can be resized via SendMessage as shown in this example: SendMessage(0x101E, 0, 80, "SysListView321", WinTitle) ; 0x101E is the message LVM_SETCOLUMNWIDTH. In the above, 0 indicates the first column (specify 1 for the second, 2 for the third, etc.) Also, 80 is the new width. Replace 80 with -1 to autosize the column. Replace it with -2 to do the same but also take into account the header text width. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetItems, WinGetList, Control functions Examples Extracts the individual rows and fields out of a ListView. List := ListViewGetContent("Selected", "SysListView321", WinTitle) Loop Parse, List, "`n" ; Rows are delimited by linefeeds (`n). { RowNumber := A_Index Loop Parse, A_LoopField, A_Tab ; Fields (columns) in each row are delimited by tabs (A_Tab). MsgBox "Row #" RowNumber " Col #" A_Index " is " A_LoopField } LoadPicture - Syntax & Usage | AutoHotkey v2 LoadPicture Loads a picture from file and returns a bitmap or icon handle. Handle := LoadPicture (Filename , Options, &ImageType) Parameters Filename Type: String The filename of the picture, which is usually assumed to be in A_WorkingDir if an absolute path isn't specified. If the name of a DLL or EXE file is given without a path, it may be loaded from the directory of the current executable (AutoHotkey.exe or a compiled script) or a system directory. Options Type: String A string of zero or more of the following options, with each separated from the last by a space or tab: Wn and Hn: The width and height to load the image at, where n is an integer. If one dimension is omitted or -1, it is calculated automatically based on the other dimension, preserving aspect ratio. If both are omitted, the image's original size is used. If either dimension is 0, the original size is used for that dimension. For example: "w80 h50", "w48 h-1" or "w48" (preserve aspect ratio), "h0 w100" (use original height but override width). Iconn: Indicates which icon to load from a file with multiple icons (generally an EXE or DLL file). For example, "Icon2" loads the file's second icon. Any supported image format can be converted to an icon by specifying "Icon1". However, the icon is converted back to a bitmap if the ImageType parameter is omitted. GDI+: Use GDI+ to load the image, if available. For example, "GDI+ w100". &ImageType Type: VarRef A reference to a output variable in which to store a number indicating the type of handle being returned: 0 (IMAGE_BITMAP), 1 (IMAGE_ICON) or 2 (IMAGE_CURSOR). If this parameter is omitted, the return value is always a bitmap handle (icons/cursors are converted if necessary). This is because reliably using or deleting an icon/cursor/bitmap handle requires knowing which type it is. Return Value Type: Integer This function returns a bitmap or icon handle depending on whether a picture or icon is specified. Remarks LoadPicture also supports the handle syntax, such as for creating a resized image based on an icon or bitmap which has already been loaded into memory, or converting an icon to a bitmap by omitting ImageType. If the image needs to be freed from memory, call whichever function is appropriate for the type of handle. if (not ImageType) ; IMAGE_BITMAP (0) or the ImageType parameter was omitted. DllCall("DeleteObject", "ptr", Handle) else if (ImageType = 1) ; IMAGE_ICON DllCall("DestroyIcon", "ptr", Handle) else if (ImageType = 2) ; IMAGE_CURSOR DllCall("DestroyCursor", "ptr", Handle) Related Image Handles Examples Pre-loads and reuses some images. Pics := [] ; Find some pictures to display. Loop Files, A_WinDir "\Web\Wallpaper*.jpg", "R" { ; Load each picture and add it to the array. Pics.Push(LoadPicture (A_LoopFileFullPath)) } if !Pics.Length { ; If this happens, edit the path on the Loop line above. MsgBox("No pictures found! Try a different directory.") ExitApp } ; Add the picture control, preserving the aspect ratio of the first picture. MyGui := Gui() Pic := MyGui.Add("Pic", "w600 h-1 +Border", "HBITMAP:*" Pics[1]) MyGui.OnEvent("Escape", (*) => ExitApp()) MyGui.OnEvent("Close", (*) => ExitApp()) MyGui.Show Loop { ; Switch pictures! Pic.Value := "HBITMAP:*" Pics[Mod(A_Index, Pics.Length)+1] Sleep 3000 } Loop - Syntax & Usage | AutoHotkey v2 Loop (normal) Performs a series of functions repeatedly: either the specified number of times or until break is encountered. Loop Count Parameters Count Type: Integer How many times (iterations) to perform the loop. If omitted, the Loop continues indefinitely until a break or return is encountered. However, an explicit blank value or number less than 1 causes the loop to be skipped entirely. Count is evaluated only once, right before the loop begins. For instance, if Count is an expression with side-effects such as function calls or assignments, the side-effects occur only once. If Count is enclosed in parentheses, a space or tab is not required. For example: Loop(2) Remarks The loop statement is usually followed by a block, which is a collection of statements that form the body of the loop. However, a loop with only a single statement does not require a block (an "if" and its "else" count as a single statement for this purpose). A common use of this statement is an infinite loop that uses the break statement somewhere in the loop's body to determine when to stop the loop. The use of break and continue inside a loop are encouraged as alternatives to goto, since they generally make a script more understandable and maintainable. One can also create a "While" or "Do...While/Until" loop by making the first or last statement of the loop's body an IF statement that conditionally issues the break statement, but the use of While or Loop...Until is usually preferred. 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, including file-loops and registry-loops; but A_Index contains 0 outside of a loop. A_Index can be assigned any integer value by the script. If Count is specified, changing A_Index affects the number of iterations that will be performed. For example, A_Index := 3 would make the loop statement act as though it is on the third iteration (A_Index will be 4 on the next iteration), while A_Index-- would

prevent the current iteration from being counted toward the total. The loop may optionally be followed by an Else statement, which is executed if Count is zero. The One True Brace (OTB) style may optionally be used. For example: Loop { ... } Loop RepeatCount { ... } Specialized loops: Loops can be used to automatically retrieve files, folders, or registry items (one at a time). See file-loop and registry-loop for details. In addition, file-reading loops can operate on the entire contents of a file, one line at a time. Finally, parsing loops can operate on the individual fields contained inside a delimited string. Related Until, While-loop, For-loop, Files-and-folders loop, Registry loop, File-reading loop, Parsing loop, Break, Continue, Blocks, Else Examples Creates a loop with 3 iterations. Loop 3 { MsgBox "Iteration number is " A_Index ; A_Index will be 1, 2, then 3 Sleep 100 } Creates an infinite loop, but it will be terminated after the 25th iteration. Loop { if (A_Index > 25) break ; Terminate the loop if (A_Index < 20) continue ; Skip the below and start a new iteration MsgBox "A_Index = " A_Index ; This will display only the numbers 20 through 25 } Loop Files - Syntax & Usage | AutoHotkey v2 Loop Files Retrieves the specified files or folders, one at a time. Loop Files FilePattern , Mode Parameters FilePattern Type: String The name of a single file or folder, or a wildcard pattern such as "C:\Temp*.tmp". FilePattern is assumed to be in A_WorkingDir if an absolute path isn't specified. Both asterisks and question marks are supported as wildcards. A match occurs when the pattern appears in either the file's long/normal name or its 8.3 short name. If this parameter is a single file or folder (i.e. no wildcards) and Mode includes R, more than one match will be found if the specified file name appears in more than one of the folders being searched. Patterns longer than 259 characters may fail to find any files due to system limitations (MAX_PATH). This limit can be bypassed by using the \\?\ long path prefix, with some stipulations. Mode Type: String If blank or omitted, only files are included and subdirectories are not recursed into. Otherwise, specify one or more of the following letters: D = Include directories (folders). F = Include files. If both F and D are omitted, files are included but not folders. R = Recurse into subdirectories (subfolders). All subfolders will be recursed into, not just those whose names match FilePattern. If R is omitted, files and folders in subfolders are not included. Special Variables Available Inside a File-Loop The following variables exist within any file-loop. If an inner file-loop is enclosed by an outer file-loop, the innermost loop's file will take precedence: Variable Description A_LoopFileName The name of the file or folder currently retrieved (without the path). A_LoopFileExt The file's extension (e.g. TXT, DOC, or EXE). The period (.) is not included. A_LoopFilePath The path and name of the file/folder currently retrieved. If FilePattern contains a relative path rather than an absolute path, the path here will also be relative. In addition, any short (8.3) folder names in FilePattern will still be short (see next item to get the long version). A_LoopFileFullPath This is different than A_LoopFilePath in the following ways: 1) It always contains the absolute/complete path of the file even if FilePattern contains a relative path; 2) Any short (8.3) folder names in FilePattern itself are converted to their long names; 3) Characters in FilePattern are converted to uppercase or lowercase to match the case stored in the file system. This is useful for converting file names -- such as those passed into a script as command line parameters -- to their exact path names as shown by Explorer. A_LoopFileShortPath The 8.3 short path and name of the file/folder currently retrieved. For example: C:\MYDOCU~1\ADDRES~1.txt. If FilePattern contains a relative path rather than an absolute path, the path here will also be relative. To retrieve the complete 8.3 path and name for a single file or folder, specify its name for FilePattern as in this example: Loop Files, "C:\My Documents\Address List.txt" ShortPathName := A_LoopFileShortPath Note: This variable will be blank if the file does not have a short name, which can happen on systems where NtfsDisable8dot3NameCreation has been set in the registry. It will also be blank if FilePattern contains a relative path and the body of the loop uses SetWorkingDir to switch away from the working directory in effect for the loop itself. A_LoopFileShortName The 8.3 short name, or alternate name of the file. If the file doesn't have one (due to the long name being shorter than 8.3 or perhaps because short-name generation is disabled on an NTFS file system), A_LoopFileName will be retrieved instead. A_LoopFileDir The path of the directory in which A_LoopFileName resides. If FilePattern contains a relative path rather than an absolute path, the path here will also be relative. A root directory will not contain a trailing backslash. For example: C: A_LoopFileTimeModified The time the file was last modified. Format YYYYMMDDHH24MISS. A_LoopFileTimeCreated The time the file was created. Format YYYYMMDDHH24MISS. A_LoopFileTimeAccessed The time the file was last accessed. Format YYYYMMDDHH24MISS. A_LoopFileAttrib The attributes of the file currently retrieved. A_LoopFileSize The size in bytes of the file currently retrieved. Files larger than 4 gigabytes are also supported. A_LoopFileSizeKB The size in Kbytes of the file currently retrieved, rounded down to the nearest integer. A_LoopFileSizeMB The size in Mbytes of the file currently retrieved, rounded down to the nearest integer. Remarks A file-loop is useful when you want to operate on a collection of files and/or folders, one at a time. All matching files are retrieved, including hidden files. By contrast, OS features such as the DIR command omit hidden files by default. To avoid processing hidden, system, and/or read-only files, use something like the following inside the loop: if A_LoopFileAttrib ~= "[HRS]" ; Skip any file that is either H (Hidden), R (Read-only), or S (System). See ~= operator. continue ; Skip this file and move on to the next one. To retrieve files' relative paths instead of absolute paths during a recursive search, use SetWorkingDir to change to the base folder prior to the loop, and then omit the path from the Loop (e.g. Loop Files, "*.*", "R"). That will cause A_LoopFilePath to contain the file's path relative to the base folder. A file-loop can disrupt itself if it creates or renames files or folders within its own purview. For example, if it renames files via FileMove or other means, each such file might be found twice: once as its old name and again as its new name. To work around this, rename the files only after creating a list of them. For example: FileList := "" Loop Files, "*.jpg" FileList .= A_LoopFileName "`n" Loop Parse, FileList, "`n" FileMove A_LoopField, "renamed_" A_LoopField Files in an NTFS file system are probably always retrieved in alphabetical order. Files in other file systems are retrieved in no particular order. To ensure a particular ordering, use the Sort function as shown in the Examples section below. File patterns longer than 259 characters are supported only when at least one of the following is true: The system has long path support enabled (requires Windows 10 version 1607 or later). The \\?\ long path prefix is used (caveats apply). In all other cases, file patterns longer than 259 characters will not find any files or folders. This limit applies both to FilePattern and any temporary pattern used during recursion into a subfolder. The One True Brace (OTB) style may optionally be used, which allows the open-brace to appear on the same line rather than underneath. For example: Loop Files "*.txt", "R" {. See Loop for information about Blocks, Break, Continue, and the A_Index variable (which exists in every type of loop). The loop may optionally be followed by an Else statement, which is executed if no matching files or directories were found (i.e. the loop had zero iterations). The functions FileGetAttrib, FileGetSize, FileGetTime, FileGetVersion, FileSetAttrib, and FileSetTime can be used in a file-loop without their Filename/FilePattern parameter. Related Loop, Break, Continue, Blocks, SplitPath, FileSetAttrib, FileSetTime Examples Reports the full path of each text file located in a directory and in its subdirectories. Loop Files, A_ProgramFiles "*.txt", "R" ; Recurse into subfolders. { Result := MsgBox("Filename = " A_LoopFilePath "`n`nContinue?",, "y/n") if Result = "No" break } Calculates the size of a folder, including the files in all its subfolders. FolderSizeKB := 0 WhichFolder := DirSelect() ; Ask the user to pick a folder. Loop Files, WhichFolder "*.*", "R" FolderSizeKB += A_LoopFileSizeKB MsgBox "Size of " WhichFolder " is " FolderSizeKB " KB." Retrieves file names sorted by name (see next example to sort by date). FileList := "" ; Initialize to be blank. Loop Files, "C:*.*" FileList .= A_LoopFileName "`n" FileList := Sort(FileList, "R") ; The R option sorts in reverse order. See Sort for other options. Loop Parse, FileList, "`n" { if A_LoopField = "" ; Ignore the blank item at the end of the list. continue Result := MsgBox("File number " A_Index " is " A_LoopField ". Continue?",, "y/n") if Result = "No" break } Retrieves file names sorted by modification date. FileList := "" Loop Files, A_MyDocuments "\Photos*.*", "FD" ; Include Files and Directories FileList .= A_LoopFileTimeModified "`t" A_LoopFileName "`n" FileList := Sort(FileList) ; Sort by date. Loop Parse, FileList, "`n" { if A_LoopField = "" ; Omit the last linefeed (blank item) at the end of the list. continue FileItem := StrSplit(A_LoopField, A_Tab) ; Split into two parts at the tab char. Result := MsgBox("The next file (modified at " FileItem[1] ") is:`n" FileItem[2] "`n`nContinue?",, "y/n") if Result = "No" break } Copies only the source files that are newer than their counterparts in the destination. Call this function with a source pattern like "A:\Scripts*.ahk" and an existing destination directory like "B:\Script Backup". CopyIfNewer(SourcePattern, Dest) { Loop Files, SourcePattern { copy_it := false if !FileExist(Dest "\" A_LoopFileName) ; Always copy if target file doesn't yet exist. copy_it := true else { time := FileGetTime(Dest "\" A_LoopFileName) time := DateDiff(time, A_LoopFileTimeModified, "Seconds") ; Subtract the source file's time from the destination's. if time < 0 ; Source file is newer than destination file. copy_it := true } if copy_it { try FileCopy A_LoopFilePath, Dest "\" A_LoopFileName, 1 ; Copy with overwrite=yes catch MsgBox 'Could not copy "' A_LoopFilePath '" to "' Dest '\' A_LoopFileName '".' } } } Converts filenames passed in via command-line parameters to long names, complete path, and correct uppercase/lowercase characters as stored in the file system. for GivenPath in A_Args ; For each parameter (or file dropped onto a script): { Loop Files, GivenPath, "FD" ; Include files and directories. LongPath := A_LoopFilePath MsgBox "The case-corrected long path name of file`n" GivenPath "`nis:`n" LongPath } Loop Parse - Syntax & Usage | AutoHotkey v2 Loop Parse Retrieves substrings (fields) from a string, one at a time. Loop Parse String , Delimiters, OmitChars Parameters String Type: String The string to analyze. Delimiters Type: String If this parameter is blank or omitted, each character of the input string will be treated as a separate substring. If this parameter is "CSV", the string will be parsed in standard comma separated value format. Here is an example of a CSV line produced by MS Excel: "first field",SecondField,"the word ""special"" is quoted literally",,"last field, has literal comma" Otherwise, Delimiters contains one or more characters (case sensitive), each of which is used to determine where the boundaries between substrings occur. Delimiter characters are not considered to be part of the substrings themselves. In addition, if there is nothing between a pair of delimiters within the input string, the corresponding substring will be empty. For example: ',' (a comma) would divide the string based on every occurrence of a comma. Similarly, A_Tab A_Space would start a new substring every time a space or tab is encountered in the input string. To use a string as a delimiter rather than a character, first use StrReplace to replace all occurrences of the string with a single character that is never used literally in the text, e.g. one of these special characters: ¢¤¥¦§©ª«®µ¶. Consider this example, which uses the string as a delimiter: NewHTML := StrReplace(HTMLString, "
", "¢") Loop Parse, NewHTML, "¢" ; Parse the string based on the cent symbol. { ... } OmitChars Type: String An optional list of characters (case sensitive) to exclude from the beginning and end of each substring. For example, if OmitChars is A_Space A_Tab, spaces and tabs will be removed from the beginning and end (but not the middle) of every retrieved substring. If Delimiters is blank, OmitChars indicates which characters should be excluded from consideration (the loop will not see them). Remarks A string parsing loop is useful when you want to operate on each field contained in a string, one at a time. Parsing loops use less memory than StrSplit (though either way the memory use is temporary) and in most cases they are easier to use. The built-in variable A_LoopField exists within any parsing

loop. It contains the contents of the current substring (field). If an inner parsing loop is enclosed by an outer parsing loop, the innermost loop's field will take precedence. Although there is no built-in variable "A_LoopDelimiter", the example at the very bottom of this page demonstrates how to detect which delimiter was encountered for each field. There is no restriction on the size of the input string or its fields. To arrange the fields in a different order prior to parsing, use the Sort function. See Loop for information about Blocks, Break, Continue, and the A_Index variable (which exists in every type of loop). The loop may optionally be followed by an Else statement, which is executed if the loop had zero iterations. Note that the loop always has at least one iteration unless String is empty or Delimiters is omitted and all characters in String are included in OmitChars. Related StrSplit, file-reading loop, Loop, Break, Continue, Blocks, Sort, FileSetAttrib, FileSetTime Examples Parses a comma-separated string. Colors := "red,green,blue" Loop parse, Colors, "," { MsgBox "Color number " A_Index " is " A_LoopField } Reads the lines inside a variable, one by one (similar to a file-reading loop). A file can be loaded into a variable via FileRead. Loop parse, FileContents, "`n", "`r" ; Specifying `n prior to `r allows both Windows and Unix files to be parsed. { Result := MsgBox("Line number " A_Index " is " A_LoopField ".`n`nContinue?",, "y/n") } until Result = "No" This is the same as the example above except that it's for the clipboard. It's useful whenever the clipboard contains files, such as those copied from an open Explorer window (the program automatically converts such files to their file names). Loop parse, A_Clipboard, "`n", "`r" { Result := MsgBox("File number " A_Index " is " A_LoopField ".`n`nContinue?",, "y/n") } until Result = "No" Parses a comma separated value (CSV) file. Loop read, "C:\Database Export.csv" { LineNumber := A_Index Loop parse, A_LoopReadLine, "CSV" { Result := MsgBox("Field " LineNumber "-" A_Index " is:`n" A_LoopField "`n`nContinue?",, "y/n") if Result = "No" return } } Determines which delimiter was encountered. ; Initialize string to search. Colors := "red,green|blue;yellow|cyan,magenta" ; Initialize counter to keep track of our position in the string. Position := 0 Loop Parse, Colors, ",|;" { ; Calculate the position of the delimiter at the end of this field. Position += StrLen(A_LoopField) + 1 ; Retrieve the delimiter found by the parsing loop. Delimiter := SubStr(Colors, Position, 1) MsgBox "Field: " A_LoopField "`nDelimiter: " Delimiter } Loop Read - Syntax & Usage | AutoHotkey v2 Loop Read Retrieves the lines in a text file, one at a time. Loop Read InputFile , OutputFile Parameters InputFile Type: String The name of the text file whose contents will be read by the loop, which is assumed to be in A_WorkingDir if an absolute path isn't specified. The file's lines may end in carriage return and linefeed (`r`n), just linefeed (`n), or just carriage return (`r). OutputFile Type: String The name of the file to be kept open for the duration of the loop, which is assumed to be in A_WorkingDir if an absolute path isn't specified. Within the loop's body, use the FileAppend function without the Filename parameter (i.e. omit it) to append to this special file. Appending to a file in this manner performs better than using FileAppend in its 2-parameter mode because the file does not need to be closed and re-opened for each operation. Remember to include a linefeed (`n) or carriage return and linefeed (`r`n) after the text, if desired. The file is not opened if nothing is ever written to it. This happens if the Loop performs zero iterations or if it never calls FileAppend. Options: The end of line (EOL) translation mode and output file encoding depend on which options are passed in the opening call to FileAppend (i.e. the first call which omits Filename). Subsequent calls ignore the Options parameter. EOL translation is not performed by default; that is, linefeed (`n) characters are written as-is unless the "`n" option is present. Standard Output (stdout): Specifying an asterisk (*) for OutputFile sends any text written by FileAppend to standard output (stdout). Such text can be redirected to a file, piped to another EXE, or captured by fancy text editors. However, text sent to stdout will not appear at the command prompt it was launched from. This can be worked around by 1) compiling the script with the Ahk2Exe ConsoleApp directive, or 2) piping a script's output to another command or program. See FileAppend for more details. Remarks A file-reading loop is useful when you want to operate on each line contained in a text file, one at a time. The file is kept open for the entire operation to avoid having to re-scan each time to find the next line. The built-in variable A_LoopReadLine exists within any file-reading loop. It contains the contents of the current line excluding the carriage return and linefeed (`r`n) that marks the end of the line. If an inner file-reading loop is enclosed by an outer file-reading loop, the innermost loop's file-line will take precedence. Lines up to 65,534 characters long can be read. If the length of a line exceeds this, its remaining characters will be read during the next loop iteration. StrSplit or a parsing loop is often used inside a file-reading loop to parse the contents of each line retrieved from InputFile. For example, if InputFile's lines are each a series of tab-delimited fields, those fields can individually retrieved as in this example: Loop read, "C:\Database Export.txt" { Loop parse, A_LoopReadLine, A_Tab { MsgBox "Field number " A_Index " is " A_LoopField "." } } To load an entire file into a variable, use FileRead because it performs much better than a loop (especially for large files). To have multiple files open simultaneously, use FileOpen. The One True Brace (OTB) style may optionally be used, which allows the open-brace to appear on the same line rather than underneath. For example: Loop Read InputFile, OutputFile {. See Loop for information about Blocks, Break, Continue, and the A_Index variable (which exists in every type of loop). To control how the file is decoded when no byte order mark is present, use FileEncoding. The loop may optionally be followed by an Else statement, which is executed if the input file is empty or could not be found. If OutputFile was specified, the special mode of FileAppend described above may also be used within the Else statement's body. If there is no Else, an OSError is thrown if the file could not be found. Related FileEncoding, FileOpen/File Object, FileRead, FileAppend, Sort, Loop, Break, Continue, Blocks, FileSetAttrib, FileSetTime Examples Only those lines of the 1st file that contain the word FAMILY will be written to the 2nd file. Uncomment the first line to overwrite rather than append to any existing file. ;FileDelete "C:\Docs\Family Addresses.txt" Loop read, "C:\Docs\Address List.txt", "C:\Docs\Family Addresses.txt" { if InStr(A_LoopReadLine, "family") FileAppend(A_LoopReadLine "`n") } else MsgBox "Address List.txt was completely empty or not found." Retrieves the last line from a text file. Loop read, "C:\Log File.txt" last_line := A_LoopReadLine ; When loop finishes, this will hold the last line. Attempts to extract all FTP and HTTP URLs from a text or HTML file. SourceFile := FileSelect(3,, "Pick a text or HTML file to analyze.") if SourceFile = "" return ; This will exit in this case. SplitPath SourceFile,, &SourceFilePath,, &SourceFileNoExt DestFile := SourceFilePath "\" SourceFileNoExt " Extracted Links.txt" if FileExist (DestFile) { Result := MsgBox("Overwrite the existing links file? Press No to append to it.`n`nFILE: " DestFile,, 4) if Result = "Yes" FileDelete DestFile } LinkCount := 0 Loop read, SourceFile, DestFile { URLSearch(A_LoopReadLine) } MsgBox LinkCount ' links were found and written to "' DestFile '".' return URLSearch(URLSearchString) { ; It's done this particular way because some URLs have other URLs embedded inside them: ; Find the left-most starting position: URLStart := 0 ; Set starting default. for URLPrefix in ["https://", "http://", "ftp://", "www."] { ThisPos := InStr(URLSearchString, URLPrefix) if !ThisPos ; This prefix is disqualified. continue if !URLStart URLStart := ThisPos else ; URLStart has a valid position in it, so compare it with ThisPos. { if ThisPos && ThisPos < URLStart URLStart := ThisPos } } if !URLStart ; No URLs exist in URLSearchString. return ; Otherwise, extract this URL: URL := SubStr(URLSearchString, URLStart) ; Omit the beginning/irrelevant part. Loop parse, URL, " `t<>" ; Find the first space, tab, or angle bracket (if any). { URL := A_LoopField break ; i.e. perform only one loop iteration to fetch the first "field". } ; If the above loop had zero iterations because there were no ending characters found, ; leave the contents of the URL var untouched. ; If the URL ends in a double quote, remove it. For now, StrReplace is used, but ; note that it seems that double quotes can legitimately exist inside URLs, so this ; might damage them: URLCleansed := StrReplace(URL, '"') FileAppend URLCleansed "`n" global LinkCount += 1 ; See if there are any other URLs in this line: CharactersToOmit := StrLen(URL) CharactersToOmit += URLStart URLSearchString := SubStr(URLSearchString, CharactersToOmit) ; Recursive call to self: URLSearch(URLSearchString) } Loop Reg - Syntax & Usage | AutoHotkey v2 Loop Reg Retrieves the contents of the specified registry subkey, one item at a time. Loop Reg KeyName , Mode Parameters KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a backslash, as in this example: \\workstation01 \HKEY_LOCAL_MACHINE Mode Type: String If blank or omitted, only values are included and subkeys are not recursed into. Otherwise, specify one or more of the following letters: K = Include keys. V = Include values. Values are also included if both K and V are omitted. R = Recurse into subkeys. If R is omitted, keys and values within subkeys of KeyName are not included. Remarks A registry-loop is useful when you want to operate on a collection registry values or subkeys, one at a time. The values and subkeys are retrieved in reverse order (bottom to top) so that RegDelete and RegDeleteKey can be used inside the loop without disrupting the loop. The following variables exist within any registry-loop. If an inner registry-loop is enclosed by an outer registry-loop, the innermost loop's registry item will take precedence: Variable Description A_LoopRegName Name of the currently retrieved item, which can be either a value name or the name of a subkey. Value names displayed by Windows RegEdit as "(Default)" will be retrieved if a value has been assigned to them, but A_LoopRegName will be blank for them. A_LoopRegType The type of the currently retrieved item, which is one of the following words: KEY (i.e. the currently retrieved item is a subkey not a value), REG_SZ, REG_EXPAND_SZ, REG_MULTI_SZ, REG_DWORD, REG_QWORD, REG_BINARY, REG_LINK, REG_RESOURCE_LIST, REG_FULL_RESOURCE_DESCRIPTOR, REG_RESOURCE_REQUIREMENTS_LIST, REG_DWORD_BIG_ENDIAN (probably rare on most Windows hardware). It will be empty if the currently retrieved item is of an unknown type. A_LoopRegKey The full name of the key which contains the current loop item. For remote registry access, this value will not include the computer name. A_LoopRegTimeModified The time the current subkey or any of its values was last modified. Format YYYYMMDDHH24MISS. This variable will be empty if the currently retrieved item is not a subkey (i.e. A_LoopRegType is not the word KEY). When used inside a registry-loop, the following functions can be used in a simplified way to indicate that the currently retrieved item should be operated upon: Syntax Description Value := RegRead() Reads the current item. If the current item is a key, an exception is thrown. RegWrite ValueRegWrite Writes to the current item. If Value is omitted, the item will be made 0 or blank depending on its type. If the current item is a key, an exception is thrown and the registry is not modified. RegDelete Deletes the current item if it is a value. If the current item is a key, its default value will be deleted instead. RegDeleteKey Deletes the current item if it is a key. If the current item is a value, the key which contains that value will be deleted, including all subkeys and values. RegCreateKey Targets a key as described above for RegDeleteKey. If the key is deleted during the loop, RegCreateKey can be used to recreate it. Otherwise, RegCreateKey merely verifies that the script has write access to the key. When accessing a remote registry (via the KeyName parameter described above), the following notes apply: The target machine must be running the Remote Registry service. Access to a remote registry may fail if the target computer is not in the same domain as yours or the local or remote username lacks sufficient permissions (however, see below for possible workarounds). Depending on your username's domain, workgroup, and/or permissions, you may have

to connect to a shared device, such as by mapping a drive, prior to attempting remote registry access. Making such a connection -- using a remote username and password that has permission to access or edit the registry -- may as a side-effect enable remote registry access. If you're already connected to the target computer as a different user (for example, a mapped drive via user Guest), you may have to terminate that connection to allow the remote registry feature to reconnect and re-authenticate you as your own currently logged-on username. The One True Brace (OTB) style may optionally be used, which allows the open-brace to appear on the same line rather than underneath. For example: Loop Reg "HKLM\Software\AutoHotkey", "V" {. See Loop for information about Blocks, Break, Continue, and the A_Index variable (which exists in every type of loop). The loop may optionally be followed by an Else statement, which is executed if no registry items of the specified type were found (i.e. the loop had zero iterations). Related Loop, Break, Continue, Blocks, RegRead, RegWrite, RegDelete, RegDeleteKey, SetRegView Examples Deletes Internet Explorer's history of URLs typed by the user. Loop Reg, "HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\TypedURLs" RegDelete A working test script. Loop Reg, "HKCU\Software\Microsoft\Windows", "R KV" ; Recursively retrieve keys and values. { if A_LoopRegType = "key" value := "" else { try value := RegRead() catch value := "*error*" } Result := MsgBox(A_LoopRegName " = " value " (" A_LoopRegType ")`n`nContinue?"„ "y/n") } Until Result = "No" Recursively searches the entire registry for particular value(s). RegSearch("Notepad") RegSearch(Target) { Loop Reg, "HKEY_LOCAL_MACHINE", "KVR" { if !CheckThisRegItem() ; It told us to stop. return } Loop Reg, "HKEY_USERS", "KVR" { if !CheckThisRegItem() ; It told us to stop. return } Loop Reg, "HKEY_CURRENT_CONFIG", "KVR" { if !CheckThisRegItem() ; It told us to stop. return } ; Note: I believe HKEY_CURRENT_USER does not need to be searched if HKEY_USERS is ; being searched. Similarly, HKEY_CLASSES_ROOT provides a combined view of keys from ; HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER, so searching all three isn't necessary. CheckThisRegItem() { if A_LoopRegType = "KEY" ; Remove these two lines if you want to check key names too. return true try RegValue := RegRead() catch return true if InStr (RegValue, Target) { Result := MsgBox(("The following match was found: " A_LoopRegKey "\" A_LoopRegName " Value = " RegValue " Continue?")„ "y/n") if Result = "No" return false ; Tell our caller to stop searching. } return true } } Map Object - Methods & Properties | AutoHotkey v2 Map Object class Map extends Object A Map object associates or maps one set of values, called keys, to another set of values. A key and the value it is mapped to are known as a key-value pair. A map can contain any number of key-value pairs, but each key must be unique. A key may be any Integer, object reference or null-terminated String. Comparison of string keys is case-sensitive, while objects are compared by reference/address. Float keys are automatically converted to String. The simplest use of a map is to retrieve or set a key-value pair via the implicit __Item property, by simply writing the key between brackets following the map object. For example: clrs := Map() clrs["Red"] := "ff0000" clrs["Green"] := "00ff00" clrs["Blue"] := "0000ff" for clr in Array("Blue", "Green") MsgBox clrs[clr] "MapObj" is used below as a placeholder for any Map object, as "Map" is the class itself. In addition to the methods and property inherited from Object, Map objects have the following predefined methods and properties. Table of Contents Static Methods: Call: Creates a Map and sets items. Methods: Clear: Removes all key-value pairs from a map. Clone: Returns a shallow copy of a map. Delete: Removes a key-value pair from a map. Get: Returns the value associated with a key, or a default value. Has: Returns true if the specified key has an associated value within a map. Set: Sets zero or more items. __Enum: Enumerates key-value pairs. __New: Sets items. Equivalent to Set. Properties: Count: Retrieves the number of key-value pairs present in a map. Capacity: Retrieves or sets the current capacity of a map. CaseSense: Retrieves or sets a map's case sensitivity setting. Default: Defines the default value returned when a key is not found. __Item: Retrieves or sets the value of a key-value pair. Static Methods Call Creates a Map and sets items. MapObj := Map(Key1, Value1, Key2, Value2, ...) This is equivalent to setting each item with MapObj[Key] := Value, except that __Item is not called and Capacity is automatically adjusted to avoid expanding multiple times during a single call. Parameters are defined by __New. Methods Clear Removes all key-value pairs from a map. MapObj.Clear() Clone Returns a shallow copy of a map. Clone := MapObj.Clone() All key-value pairs are copied to the new map. Object references are copied (like with a normal assignment), not the objects themselves. Own properties, own methods and base are copied as per Obj.Clone. Delete Removes a key-value pair from a map. MapObj.Delete(Key) Key Type: Integer, object or String Any single key. If the map does not contain this key, an UnsetItemError is thrown. Returns the removed value. Get Returns the value associated with a key, or a default value. Value := MapObj.Get(Key , Default) This method does the following: Return the value associated with Key, if found. Return the value of the Default parameter, if specified. Return the value of MapObj.Default, if defined. Throw an UnsetItemError. When Default is omitted, this is equivalent to MapObj[Key], except that __Item is not called. Has Returns true if the specified key has an associated value within a map, otherwise false. MapObj.Has(Key) Set Sets zero or more items. MapObj.Set(Key, Value, Key2, Value2, ...) This is equivalent to setting each item with MapObj[Key] := Value, except that __Item is not called and Capacity is automatically adjusted to avoid expanding multiple times during a single call. Returns the map. __Enum Enumerates key-value pairs. For Key , Value in MapObj Returns a new enumerator. This method is typically not called directly. Instead, the map object is passed directly to a for-loop, which calls __Enum once and then calls the enumerator once for each iteration of the loop. Each call to the enumerator returns the next key and/or value. The for-loop's variables correspond to the enumerator's parameters, which are: Key Type: Integer, object or String The key. Value The value. __New Sets items. Equivalent to Set. MapObj.__New(Key, Value, Key2, Value2, ...) This method exists to support Call, and is not intended to be called directly. See Construction and Destruction. Properties Count Retrieves the number of key-value pairs present in a map. Count := MapObj.Count Capacity Retrieves or sets the current capacity of a map. MapObj.Capacity := MaxItems MaxItems := MapObj.Capacity MaxItems Type: Integer The maximum number of key-value pairs the map should be able to contain before it must be automatically expanded. If setting a value less than the current number of key-value pairs, that number is used instead, and any unused space is freed. CaseSense Retrieves or sets a map's case sensitivity setting. MapObj.CaseSense := Setting Setting := MapObj.CaseSense Setting Type: String One of the following values: "On": Key lookups are case sensitive. This is the default setting. "Off": The letters A-Z are considered identical to their lowercase counterparts. "Locale": Key lookups are case insensitive according to the rules of the current user's locale. For example, most English and Western European locales treat not only the letters A-Z as identical to their lowercase counterparts, but also non-ASCII letters like Ä and Ü as identical to theirs. Locale is 1 to 8 times slower than Off depending on the nature of the strings being compared. When assigning a value, the strings "1" and "0" can also be used. Attempting to assign to this property causes an exception to be thrown if the Map is not empty. Default Defines the default value returned when a key is not found. MapObj.Default := Value This property actually doesn't exist by default, but can be defined by the script. When the script queries a key which is not found, the map checks for this property before throwing an UnsetItemError. It can be implemented by any of the normal means, including a dynamic property or meta-function, but determining which key was queried would require overriding __Item or Get instead. __Item Retrieves or sets the value of a key-value pair. Value := MapObj[Key] MapObj[Key] := Value Key Type: Integer, object or String When retrieving a value, Key must be a unique value previously associated with another value. An UnsetItemError is thrown if Key has no associated value within the map, unless a Default property is defined, in which case its value is returned. When assigning a value, Key can be any value to associate with Value; in other words, the key used to later access Value. Float keys are automatically converted to String. The property name __Item is typically omitted, as shown above, but is used when overriding the property. Math Functions - Syntax & Usage | AutoHotkey v2 Math Functions Functions for performing various mathematical operations such as rounding, exponentiation, squaring, etc. Table of Contents General Math: Abs: Returns the absolute value of a number. Ceil: Returns a number rounded up to the nearest integer. Exp: Returns e raised to the Nth power. Floor: Returns a number rounded down to the nearest integer. Log: Returns the logarithm (base 10) of a number. Ln: Returns the natural logarithm (base e) of a number. Max: Returns the highest value of one or more numbers. Min: Returns the lowest value of one or more numbers. Mod: Returns the remainder of a division. Round: Returns a number rounded to N decimal places. Sqrt: Returns the square root of a number. Trigonometry: Sin: Returns the trigonometric sine of a number. Cos: Returns the trigonometric cosine of a number. Tan: Returns the trigonometric tangent of a number. ASin: Returns the arcsine (the number whose sine is the specified number) in radians. ACos: Returns the arccosine (the number whose cosine is the specified number) in radians. ATan: Returns the arctangent (the number whose tangent is the specified number) in radians. Error-handling General Math Abs Returns the absolute value of the specified number. Value := Abs(Number) The return value is the same type as Number (integer or floating point). MsgBox Abs(-1.2) ; Returns 1.2 Ceil Returns the specified number rounded up to the nearest integer (without any .00 suffix). Value := Ceil(Number) MsgBox Ceil(1.2) ; Returns 2 MsgBox Ceil(-1.2) ; Returns -1 Exp Returns e (which is approximately 2.71828182845905) raised to the Nth power. Value := Exp(N) N may be negative and may contain a decimal point. To raise numbers other than e to a power, use the ** operator. MsgBox Exp(1.2) ; Returns 3.320117 Floor Returns the specified number rounded down to the nearest integer (without any .00 suffix). Value := Floor(Number) MsgBox Floor(1.2) ; Returns 1 MsgBox Floor(-1.2) ; Returns -2 Log Returns the logarithm (base 10) of the specified number. Value := Log(Number) The result is a floating-point number. If Number is negative, a ValueError is thrown. MsgBox Log(1.2) ; Returns 0.079181 Ln Returns the natural logarithm (base e) of the specified number. Value := Ln(Number) The result is a floating-point number. If Number is negative, a ValueError is thrown. MsgBox Ln(1.2) ; Returns 0.182322 Max Returns the highest value of one or more numbers. Value := Max(Number1 , Number2, ...) MsgBox Max(2.11, -2, 0) ; Returns 2.11 You can also specify a variadic parameter to compare multiple values within an array. For example: Numbers := [1, 2, 3, 4] MsgBox Max(Numbers*) ; Returns 4 Min Returns the lowest value of one or more numbers. Value := Min(Number1 , Number2, ...) MsgBox Min(2.11, -2, 0) ; Returns -2 You can also specify a variadic parameter to compare multiple values within an array. For example: Numbers := [1, 2, 3, 4] MsgBox Min(Numbers*) ; Returns 1 Mod Modulo. Returns the remainder of the specified dividend divided by the specified divisor. Value := Mod (Dividend, Divisor) The sign of the result is always the same as the sign of the first parameter. If either input is a floating point number, the result is also a floating point number. If the second parameter is zero, a ZeroDivisionError is thrown. MsgBox Mod(7.5, 2) ; Returns 1.5 (2 x 3 + 1.5) Round Returns the specified number rounded to N decimal places. Value := Round(Number , N) If N is omitted or 0, Number is rounded to the nearest integer: MsgBox Round(3.14) ; Returns 3 If N is positive number, Number is rounded to N decimal places: MsgBox Round(3.14, 1) ; Returns 3.1 If N is negative, Number is rounded by N digits to the left of the decimal point: MsgBox Round(345, -1) ; Returns 350 MsgBox Round(345, -2) ; Returns 300 The result is an integer if N is omitted or less than 1. Otherwise, the result is a numeric string with exactly N decimal places. If a pure number is needed, simply perform another math operation on Round's return value; for example:

Round(3.333, 1)+0. Sqrt Returns the square root of the specified number. Value := Sqrt(Number) The result is a floating-point number. If Number is negative, a ValueError is thrown. MsgBox Sqrt(16) ; Returns 4 Trigonometry Note: To convert a radians value to degrees, multiply it by 180/pi (approximately 57.29578). To convert a degrees value to radians, multiply it by pi/180 (approximately 0.01745329252). The value of pi (approximately 3.141592653589793) is 4 times the arctangent of 1. Sin Returns the trigonometric sine of the specified number. Value := Sin(Number) Number must be expressed in radians. MsgBox Sin(1.2) ; Returns 0.932039 Cos Returns the trigonometric cosine of the specified number. Value := Cos(Number) Number must be expressed in radians. MsgBox Cos(1.2) ; Returns 0.362358 Tan Returns the trigonometric tangent of the specified number. Value := Tan(Number) Number must be expressed in radians. MsgBox Tan(1.2) ; Returns 2.572152 ASin Returns the arcsine (the number whose sine is the specified number) in radians. Value := ASin(Number) If Number is less than -1 or greater than 1, a ValueError is thrown. MsgBox ASin(0.2) ; Returns 0.201358 ACos Returns the arccosine (the number whose cosine is the specified number) in radians. Value := ACos(Number) If Number is less than -1 or greater than 1, a ValueError is thrown. MsgBox ACos(0.2) ; Returns 1.369438 ATan Returns the arctangent (the number whose tangent is the specified number) in radians. Value := ATan(Number) MsgBox ATan(1.2) ; Returns 0.876058 Error-Handling These functions throw an exception if any incoming parameters are non-numeric or an invalid operation (such as divide by zero) is attempted. Menu/MenuBar Object - Methods & Properties | AutoHotkey v2 Menu/MenuBar Object Used to modify and display menus or menu bars. class Menu extends Object Menu objects are used to define, modify and display popup menus. Menu(), MenuFromHandle and A_TrayMenu return an object of this type. class MenuBar extends Menu MenuBar objects are used to define and modify menu bars for use with Gui.MenuBar. They are created with MenuBar(). MenuFromHandle returns an object of this type if given a menu bar handle. "MyMenu" is used below as a placeholder for any Menu object, as "Menu" is the class itself. In addition to the methods and property inherited from Object, Menu objects have the following predefined methods and properties. Table of Contents Static Methods: Call: Creates a new Menu or MenuBar object. Methods: Add: Adds or modifies a menu item. AddStandard: Adds the standard tray menu items. Check: Adds a visible checkmark next to a menu item. Delete: Deletes a menu item or all menu items. Disable: Changes a menu item to a gray color to indicate that the user cannot select it. Enable: Allows the user to once again select a menu item if was previously disabled (grayed). Insert: Inserts a new item before the specified item. Rename: Renames a menu item. SetColor: Changes the background color of the menu. SetIcon: Sets the icon to be displayed next to a menu item. Show: Displays the menu. ToggleCheck: Toggles the checkmark next to a menu item. ToggleEnable: Enables or disables a menu item. Uncheck: Removes the checkmark (if there is one) from a menu item. Properties: ClickCount: Retrieves or sets the number of clicks required to activate the tray menu's default item. Default: Retrieves or sets the default menu item. Handle: Retrieves the menu's Win32 handle. General: MenuItemName Win32 Menus Remarks Related Examples Static Methods Call Creates a new Menu or MenuBar object. MyMenu := Menu() MyMenuBar := MenuBar() Methods Add Adds or modifies a menu item. MyMenu.Add(MenuItemName, Function-or-Submenu, Options) MenuItemName Type: String The text to display on the menu item, or the position& of an existing item to modify. See MenuItemName. Function-or-Submenu Type: Function Object or Menu A function object to call as a new thread when the menu item is selected, or a reference to a Menu object to use as a submenu. This parameter is required when creating a new item, but optional when updating the Options of an existing item. The function should accept the following parameters: FunctionName(ItemName, ItemPos, MyMenu) Options Type: String If not omitted, Options must be a space- or tab-delimited list of one or more of the following options: Option Description Pn Replace n with the menu item's thread priority, e.g. P1. If this option is omitted when adding a menu item, the priority will be 0, which is the standard default. If omitted when updating a menu item, the item's priority will not be changed. Use a decimal (not hexadecimal) number as the priority. +Radio If the item is checked, a bullet point is used instead of a check mark. +Right The item is right-justified within the menu bar. This only applies to menu bars, not popup menus or submenus. +Break The item begins a new column in a popup menu. +BarBreak As above, but with a dividing line between columns. The plus sign (+) is optional and can be replaced with minus (-) to remove the option, as in -Radio. Options are not case sensitive. To change an existing item's options without affecting its callback or submenu, simply omit the Callback-or-Submenu parameter. This is a multipurpose method that adds a menu item, updates one with a new submenu or callback, or converts one from a normal item into a submenu (or vice versa). If MenuItemName does not yet exist, it will be added to the menu. Otherwise, MenuItemName is updated with the newly specified Callback-or-Submenu and/or Options. To add a menu separator line, omit all three parameters. Add always adds new menu items at the bottom of the menu, but Insert can be used to insert an item before an existing custom menu item. AddStandard Adds the standard tray menu items. MyMenu.AddStandard() This method can be used with the tray menu or any other menu. The standard items are inserted after any existing items. Any standard items already in the menu are not duplicated, but any missing items are added. The table below shows the names and positions of the standard items after calling AddStandard on an empty menu: &Open10 &Help2 3 &Window Spy4 &Reload Script5 &Edit Script6 7 &Suspend Hotkeys81 &Pause Script92 E&xit103 Compiled scripts include only the last three by default. &Open is included only if A_AllowMainWindow is 1 when AddStandard is called (in that case, add 1 to the positions shown in the third column). If the tray menu contains standard items, &Open is inserted or removed whenever A_AllowMainWindow is changed. For other menus, &Open has no effect if A_AllowMainWindow is 0. Each standard item has an internal menu item ID corresponding to the function it performs, but can otherwise be modified or deleted like any other menu item. AddStandard detects existing items by ID, not by name. If the Add method is used to change the callback function associated with a standard menu item, it is assigned a new unique ID and is no longer considered to be a standard item. Adding the &Open item to the tray menu causes it to become the default item if there wasn't one already. Check Adds a visible checkmark in the menu next to MenuItemName (if there isn't one already). MyMenu.Check(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. Delete Deletes a menu item or all custom menu items. MyMenu.Delete(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. If MenuItemName is omitted, all items are deleted from the menu, leaving the menu empty. An empty menu still exists and thus any other menus that use it as a submenu will retain those submenus. To delete a separator line, identify it by its position in the menu. For example, use MyMenu.Delete("3&") if there are two items preceding the separator. If the default menu item is deleted, the effect will be similar to having set MyMenu.Default := "". Disable Changes MenuItemName to a gray color to indicate that the user cannot select it. MyMenu.Disable(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. Enable Allows the user to once again select MenuItemName if it was previously disabled (grayed). MyMenu.Enable(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. Insert Inserts a new item before the specified item. MyMenu.Insert(ItemToInsertBefore, NewItemName, Callback-or-Submenu, Options) ItemToInsertBefore Type: String The name of an existing item or a position& between 1 and the current number of custom items plus 1 (following the same rules as MenuItemName). Items can also be appended by omitting ItemToInsertBefore. NewItemName Type: String The text to display on the menu item. Unlike Add, this cannot be a position. The remaining parameters behave as per the Add method, except that Insert creates a new item even if NewItemName matches the name of an existing item. Rename Renames MenuItemName to NewName. MyMenu.Rename(MenuItemName , NewName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. NewName Type: String The new name. If empty or omitted, MenuItemName will be converted into a separator line. The menu item's current callback or submenu is unchanged. A separator line can be converted to a normal item by specifying the position& of the separator and a non-blank NewName, and then using the Add method to give the item a callback or submenu. SetColor Changes the background color of the menu to ColorValue. MyMenu.SetColor (ColorValue, ApplyToSubmenus) ColorValue Type: String or Integer One of the 16 primary HTML color names, a hexadecimal RGB color string (the 0x prefix is optional), or a pure numeric RGB color value. Omit ColorValue (or specify an empty string or the word "Default") to restore the menu to its default color. Example values: "Silver", "FFFFAA", 0xFFFFAA, "Default". ApplyToSubmenus Type: Boolean 1 (true) if the color should be applied to all of this menu's submenus, otherwise 0 (false). Defaults to 1 (true). SetIcon Sets the icon to be displayed next to MenuItemName. MyMenu.SetIcon(MenuItemName, FileName , IconNumber, IconWidth) MenuItemName Type: String The name or position of a menu item. See MenuItemName. FileName Type: String The path of an icon or image file. For a list of supported formats, see the Picture control. A bitmap or icon handle can be used instead of a filename. For example, "HICON:" handle. Omit FileName or specify an empty string or "*" to remove the item's current icon. IconNumber Type: Integer To use an icon group other than the first one in the file, specify its number for IconNumber (if omitted, it defaults to 1). If IconNumber is negative, its absolute value is assumed to be the resource ID of an icon within an executable file. IconWidth Type: Integer The desired width of the icon. If the icon group indicated by IconNumber contains multiple icon sizes, the closest match is used and the icon is scaled to the specified size. See the Examples section for usage examples. Currently it is necessary to specify "actual size" when setting the icon to preserve transparency. For example: MyMenu.SetIcon "My menu item", "Filename.png",, 0 A bitmap or icon handle can be used instead of a filename. For example, "HBITMAP:" handle. Show Displays the menu, allowing the user to select an item with arrow keys, menu shortcuts (underlined letters), or the mouse. MyMenu.Show(X, Y) X, Y Type: Integer The coordinates at which to display the menu. If both X and Y are omitted, the menu is displayed at the current position of the mouse cursor. If only one of them is omitted, the mouse cursor's position will be used for it. X and Y are relative to the active window's client area by default. To override this default, use CoordMode "Menu", Mode or A_CoordModeMenu := Mode. Any popup menu can be shown, including submenus and the tray menu. However, an exception is thrown if Menu is a MenuBar object. ToggleCheck Adds a checkmark if there wasn't one; otherwise, removes it. MyMenu.ToggleCheck (MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. ToggleEnable Disables MenuItemName if it was previously enabled; otherwise, enables it. MyMenu.ToggleEnable(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. Uncheck Removes the checkmark (if there is one) from a menu item. MyMenu.Uncheck(MenuItemName) MenuItemName Type: String The name or position of a menu item. See MenuItemName. Properties ClickCount Retrieves or sets the number of clicks required to activate the tray menu's default item. MyMenu.ClickCount := Count Count Type: Integer Specify 1 to allow a single-click to activate the tray menu's default menu item. Specify 2 to return to the default behavior (double-click). For example: A_TrayMenu.ClickCount := 1 Default Retrieves or sets the default menu item. CurrentDefault := MyMenu.Default CurrentDefault Type: String The name of the default menu item, or an empty string if there is no default. MyMenu.Default := MenuItemName MenuItemName

Type: String The name or position of a menu item. See MenuItemName. If MenuItemName is an empty string, there will be no default. Setting the default item makes that item's font bold (setting a default item in menus other than the tray menu is currently purely cosmetic). When the user double-clicks the tray icon, its default menu item is launched (even if the item is disabled). If there is no default, double-clicking has no effect. The default item for the tray menu is initially &Open, if present. Adding &Open to the tray menu by calling AddStandard or changing A_AllowMainWindow also causes it to become the default item if there wasn't one already. If the default item is deleted, the menu is left without one. Handle Returns a handle to a Win32 menu (a handle of type HMENU), constructing it if necessary. MyMenu.Handle The returned handle is valid only until the Win32 menu is destroyed, which typically occurs when the Menu object is freed. Once the menu is destroyed, the operating system may reassign the handle value to any menus subsequently created by the script or any other program. MenuItemName The name or position of a menu item. Some common rules apply to this parameter across all methods which use it: To underline one of the letters in a menu item's name, precede that letter with an ampersand (&). When the menu is displayed, such an item can be selected by pressing the corresponding key on the keyboard. To display a literal ampersand, specify two consecutive ampersands as in this example: "Save && Exit" When referring to an existing menu item, the name is not case sensitive but any ampersands must be included. For example: "&Open" The names of menu items can be up to 260 characters long. To identify an existing item by its position in the menu, write the item's position followed by an ampersand. For example, "1&" indicates the first item. Win32 Menus Windows provides a set of functions and notifications for creating, modifying and displaying menus with standard appearance and behavior. We refer to a menu created by one of these functions as a Win32 menu. As items are added to a menu or modified, the name and other properties of each item are stored in the Menu object. A Win32 menu is constructed the first time the menu or its parent menu is attached to a GUI or shown. It is destroyed automatically when the menu object is deleted (which occurs when its reference count reaches zero). Menu.Handle returns a handle to a Win32 menu (a handle of type HMENU), constructing it if necessary. Any modifications which are made to the menu directly by Win32 functions are not reflected by the script's Menu object, so may be lost if an item is modified by one of the built-in methods. Each menu item is assigned an ID when it is first added to the menu. Scripts cannot rely on an item receiving a particular ID, but can retrieve the ID of an item by using GetMenuItemID as shown in example #5. This ID cannot be used with the Menu object, but can be used with various Win32 functions. Remarks A menu usually looks like this: If a menu ever becomes completely empty -- such as by using MyMenu.Delete() -- it cannot be shown. If the tray menu becomes empty, right-clicking and double-clicking the tray icon will have no effect (in such cases it is usually better to use #NoTrayIcon). If a menu item's callback is already running and the user selects the same menu item again, a new thread will be created to run that same callback, interrupting the previous thread. To instead buffer such events until later, use Critical as the callback's first line (however, this will also buffer/defer other threads such as the press of a hotkey). Whenever a function is called via a menu item, it starts off fresh with the default values for settings such as SendMode. These defaults can be changed during script startup. When building a menu whose contents are not always the same, one approach is to point all such menu items to the same function and have that function refer to its parameters to determine what action to take. Alternatively, a function object, closure or fat arrow function can be used to bind one or more values or variables to the menu item's callback function. Related GUI, Threads, Thread, Critical, #NoTrayIcon, Functions, Return, SetTimer Examples Adds a new menu item to the bottom of the tray icon menu. A_TrayMenu.Add() ; Creates a separator line. A_TrayMenu.Add("Item1", MenuHandler) ; Creates a new menu item. Persistent MenuHandler(ItemName, ItemPos, MyMenu) { MsgBox "You selected " ItemName " (position " ItemPos ")" } Creates a popup menu that is displayed when the user presses a hotkey. ; Create the popup menu by adding some items to it. MyMenu := Menu() MyMenu.Add "Item 1", MenuHandler MyMenu.Add "Item 2", MenuHandler MyMenu.Add ; Add a separator line. ; Create another menu destined to become a submenu of the above menu. Submenu1 := Menu() Submenu1.Add "Item A", MenuHandler Submenu1.Add "Item B", MenuHandler ; Create a submenu in the first menu (a right-arrow indicator). When the user selects it, the second menu is displayed. MyMenu.Add "My Submenu", Submenu1 MyMenu.Add ; Add a separator line below the submenu. MyMenu.Add "Item 3", MenuHandler ; Add another menu item beneath the submenu. MenuHandler(Item, *) { MsgBox "You selected " Item } #z::MyMenu.Show ; i.e. press the Win-Z hotkey to show the menu. Demonstrates some of the various menu object members. #SingleInstance Persistent tray := A_TrayMenu ; For convenience. tray.delete ; Delete the standard items. tray.add ; separator tray.add "TestToggleCheck", TestToggleCheck tray.add "TestToggleEnable", TestToggleEnable tray.add "TestDefault", TestDefault tray.add "TestAddStandard", TestAddStandard tray.add "TestDelete", TestDelete tray.add "TestDeleteAll", TestDeleteAll tray.add "TestRename", TestRename tray.add "Test", Test ;; TestToggleCheck(*) { tray.ToggleCheck "TestToggleCheck" tray.Enable "TestToggleEnable" ; Also enables the next test since it can't undo the disabling of itself. tray.add "TestDelete", TestDelete ; Similar to above. } TestToggleEnable (*) { tray.ToggleEnable "TestToggleEnable" } TestDefault(*) { if tray.default = "TestDefault" tray.default := "" else tray.default := "TestDefault" } TestAddStandard(*) { tray.addStandard } TestDelete(*) { tray.delete "TestDelete" } TestDeleteAll(*) { tray.delete } TestRename(*) { static OldName := "", NewName := "" if NewName != "renamed" { OldName := "TestRename" NewName := "renamed" } else { OldName := "renamed" NewName := "TestRename" } tray.rename OldName, NewName } Test(Item, *) { MsgBox 'You selected "' Item '"' } Demonstrates how to add icons to menu items. FileMenu := Menu() FileMenu.Add("Script Icon", MenuHandler) FileMenu.Add("Suspend Icon", MenuHandler) FileMenu.Add("Pause Icon", MenuHandler) FileMenu.SetIcon ("Script Icon", A_AhkPath, 2) ; 2nd icon group from the file FileMenu.SetIcon("Suspend Icon", A_AhkPath, -206) ; icon with resource ID 206 FileMenu.SetIcon ("Pause Icon", A_AhkPath, -207) ; icon with resource ID 207 MyMenuBar := MenuBar() MyMenuBar.Add("&File", FileMenu) MyGui := Gui() MyGui.MenuBar := MyMenuBar MyGui.Add("Button",, "Exit This Example").OnEvent("Click", (*) => WinClose()) MyGui.Show MenuHandler(*) { ; For this example, the menu items don't do anything. } Reports the number of items in a menu and the ID of the last item. MyMenu := Menu() MyMenu.Add "Item 1", NoAction MyMenu.Add "Item 2", NoAction MyMenu.Add "Item B", NoAction ; Retrieve the number of items in a menu. item_count := DllCall ("GetMenuItemCount", "ptr", MyMenu.Handle) ; Retrieve the ID of the last item. last_id := DllCall("GetMenuItemID", "ptr", MyMenu.Handle, "int", item_count-1) MsgBox "MyMenu has " item_count " items, and its last item has ID " last_id NoAction(*) { ; Do nothing. } MenuFromHandle - Syntax & Usage | AutoHotkey v2 MenuFromHandle Retrieves the Menu or MenuBar object corresponding to a Win32 menu handle. Menu := MenuFromHandle(Handle) Parameters Handle Type: Integer A handle to a Win32 menu (of type HMENU). Remarks If the handle is invalid or does not correspond to a menu created by this script, the function returns an empty string. Related Win32 Menus, Menu/MenuBar object, Menu.Handle, Menu() MenuSelect - Syntax & Usage | AutoHotkey v2 MenuSelect Invokes a menu item from the menu bar of the specified window. MenuSelect WinTitle, WinText, Menu , SubMenu1, SubMenu2, SubMenu3, SubMenu4, SubMenu5, SubMenu6, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. Menu Type: String The name (or a prefix of the name) of the top-level menu item, e.g. File, Edit, View. It can also be the position of the desired menu item by using 1& to represent the first menu, 2& the second, and so on. The search is case-insensitive according to the rules of the current user's locale, and stops at the first matching item. The use of ampersand (&) to indicate the underlined letter in a menu item is usually not necessary (i.e. &File is the same as File). Known limitation: If the parameter contains an ampersand, it must match the item name exactly, including all non-literal ampersands (which are hidden or displayed as an underline). If the parameter does not contain an ampersand, all ampersands are ignored, including literal ones. For example, an item displayed as "a & b" may match a parameter value of a && b or a b. Specify 0& to use the window's system menu. SubMenu1 Type: String The name of the menu item to select or its position (see above). This can be omitted if the top-level item does not contain a menu (rare). SubMenu2 Type: String If SubMenu1 itself contains a menu, this is the name of the menu item inside, or its position. SubMenu3 SubMenu4 SubMenu5 SubMenu6 Type: String Same as above. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found, or does not have a standard Win32 menu. A ValueError is thrown if a menu, submenu or menu item could not be found, or if the final menu parameter corresponds to a menu item which opens a submenu. Remarks For this function to work, the target window need not be active. However, some windows might need to be in a non-minimized state. This function will not work with applications that use non-standard menu bars. Examples include Microsoft Outlook and Outlook Express, which use disguised toolbars for their menu bars. In these cases, consider using ControlSend or PostMessage, which should be able to interact with some of these non-standard menu bars. The menu name parameters can also specify positions. This method exists to support menus that don't contain text (perhaps because they contain pictures of text rather than actual text). Position 1& is the first menu item (e.g. the File menu), position 2& is the second menu item (e.g. the Edit menu), and so on. Menu separator lines count as menu items for the purpose of determining the position of a menu item. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. System Menu Menu can be 0& to select an item within the window's system menu, which typically appears when the user presses Alt+Space or clicks on the icon in the window's title bar. For example: ; Paste a command into cmd.exe without activating the window. A_Clipboard := "echo Hello, world!`r" MenuSelect "ahk_exe cmd.exe",, "0&", "Edit", "Paste" Caution: Use this only on windows which have custom items in their system menu. If the window does not already have a custom system menu, a copy of the standard system menu will be created and assigned to the target window as a side effect. This copy is destroyed by the system when the script exits, leaving other scripts unable to access it. Therefore, avoid using 0& for the standard items which appear on all windows. Instead, post the WM_SYSCOMMAND message directly. For example: ; Like [WinMinimize "A"], but also play the system sound for minimizing. WM_SYSCOMMAND := 0x0112 SC_MINIMIZE := 0xF020 PostMessage WM_SYSCOMMAND, SC_MINIMIZE, 0,, "A" Related ControlSend, PostMessage Examples Selects File -> Open in Notepad. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. MenuSelect "Untitled - Notepad",, "File", "Open" Same as above except it is done by position instead of name. On Windows 10, 2& must be replaced with 3& due to the new "New Window" menu item. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. MenuSelect "Untitled - Notepad",, "1&", "2&" Selects View -> Lines most recently executed in the main window. WinShow "ahk_class AutoHotkey" MenuSelect

"ahk_class AutoHotkey",, "View", "Lines most recently executed" List of Monitor Functions | AutoHotkey v2 Monitor Functions Functions for retrieving screen resolution and multi-monitor info. Click on a function name for details. Function Description MonitorGet Checks if the specified monitor exists and optionally retrieves its bounding coordinates. MonitorGetCount Returns the total number of monitors. MonitorGetName Returns the operating system's name of the specified monitor. MonitorGetPrimary Returns the number of the primary monitor. MonitorGetWorkArea Checks if the specified monitor exists and optionally retrieves the bounding coordinates of its working area. Remarks The built-in variables A_ScreenWidth and A_ScreenHeight contain the dimensions of the primary monitor, in pixels. SysGet can be used to retrieve the bounding rectangle of all display monitors. For example, this retrieves the width and height of the virtual screen: MsgBox SysGet(78) " x " SysGet(79) Related DllCall, Win functions, SysGet Examples Displays info about each monitor. MonitorCount := MonitorGetCount() MonitorPrimary := MonitorGetPrimary() MsgBox "Monitor Count:`t" MonitorCount "`nPrimary Monitor:`t" MonitorPrimary Loop MonitorCount { MonitorGet A_Index, &L, &T, &R, &B MonitorGetWorkArea A_Index, &WL, &WT, &WR, &WB MsgBox ( "Monitor:`t#" A_Index " Name:`t" MonitorGetName(A_Index) " Left:`t" L " (" WL " work) Top:`t" T " (" WT " work) Right:`t" R " (" WR " work) Bottom:`t" B " (" WB " work)") } MonitorGet - Syntax & Usage | AutoHotkey v2 MonitorGet Checks if the specified monitor exists and optionally retrieves its bounding coordinates. ActualN := MonitorGet(N, &Left, &Top, &Right, &Bottom) Parameters N Type: Integer The monitor number, between 1 and the number returned by MonitorGetCount. If omitted, the primary monitor is used. &Left &Top &Right &Bottom Type: VarRef References to the output variables in which to store the bounding coordinates, in pixels. Return Value Type: Integer This function returns the monitor number (the same as N unless N was omitted). Error Handling On failure, an exception is thrown and the output variables are not modified. Remarks The built-in variables A_ScreenWidth and A_ScreenHeight contain the dimensions of the primary monitor, in pixels. SysGet can be used to retrieve the bounding rectangle of all display monitors. For example, this retrieves the width and height of the virtual screen: MsgBox SysGet(78) " x " SysGet(79) Related MonitorGetWorkArea, SysGet, Monitor functions Examples Shows the bounding coordinates of the second monitor in a message box. try { MonitorGet 2, &Left, &Top, &Right, &Bottom MsgBox "Left: " Left " -- Top: " Top " -- Right: " Right " -- Bottom: " Bottom } catch MsgBox "Monitor 2 doesn't exist or an error occurred." See example #1 on the Monitor Functions page for another demonstration of this function. MonitorGetCount - Syntax & Usage | AutoHotkey v2 MonitorGetCount Returns the total number of monitors. Count := MonitorGetCount() Parameters This function has no parameters. Return Value Type: Integer This function returns the total number of monitors. Remarks Unlike the SM_CMONITORS system property retrieved via SysGet(80), the return value includes all monitors, even those not being used as part of the desktop. Related SysGet, Monitor functions Examples See example #1 on the Monitor Functions page for a demonstration of this function. MonitorGetName - Syntax & Usage | AutoHotkey v2 MonitorGetName Returns the operating system's name of the specified monitor. Name := MonitorGetName(N) Parameters N Type: Integer The monitor number, between 1 and the number returned by MonitorGetCount. If omitted, the primary monitor is used. Return Value Type: String This function returns the operating system's name for the specified monitor. Error Handling An exception is thrown on failure. Related SysGet, Monitor functions Examples See example #1 on the Monitor Functions page for a demonstration of this function. MonitorGetPrimary - Syntax & Usage | AutoHotkey v2 MonitorGetPrimary Returns the number of the primary monitor. Primary := MonitorGetPrimary() Parameters This function has no parameters. Return Value Type: Integer This function returns the number of the primary monitor. In a single-monitor system, this will be always 1. Related SysGet, Monitor functions Examples See example #1 on the Monitor Functions page for a demonstration of this function. MonitorGetWorkArea - Syntax & Usage | AutoHotkey v2 MonitorGetWorkArea Checks if the specified monitor exists and optionally retrieves the bounding coordinates of its working area. IsExisting := MonitorGetWorkArea(N, &Left, &Top, &Right, &Bottom) Parameters N Type: Integer The monitor number, between 1 and the number returned by MonitorGetCount. If omitted, the primary monitor is used. &Left &Top &Right &Bottom Type: VarRef References to the output variables in which to store the bounding coordinates of the working area, in pixels. Return Value Type: Integer This function returns the monitor number (the same as N unless N was omitted). Error Handling On failure, an exception is thrown and the output variables are not modified. Remarks The working area of a monitor excludes the area occupied by the taskbar and other registered desktop toolbars. Related MonitorGet, SysGet, Monitor functions Examples See example #1 on the Monitor Functions page for a demonstration of this function. MouseClick - Syntax & Usage | AutoHotkey v2 MouseClick Clicks or holds down a mouse button, or turns the mouse wheel. NOTE: The Click function is generally more flexible and easier to use. MouseClick WhichButton, X, Y, ClickCount, Speed, DownOrUp, Relative Parameters WhichButton Type: String The button to click: Left (default), Right, Middle (or just the first letter of each of these); or the fourth or fifth mouse button (X1 or X2). For example: MouseClick "X1". This parameter may be omitted, in which case it defaults to Left. Left and Right correspond to the primary button and secondary button. If the user swaps the buttons via system settings, the physical positions of the buttons are swapped but the effect stays the same. Rotate the mouse wheel: Specify WheelUp or WU to turn the wheel upward (away from you); specify WheelDown or WD to turn the wheel downward (toward you). Specify WheelLeft (or WL) or WheelRight (or WR) to push the wheel left or right, respectively. ClickCount is the number of notches to turn the wheel. X, Y Type: Integer The x/y coordinates to which the mouse cursor is moved prior to clicking. Coordinates are relative to the active window's client area unless CoordMode was used to change that. If omitted, the cursor's current position is used. ClickCount Type: Integer The number of times to click the mouse. If omitted, the button is clicked once. Speed Type: Integer The speed to move the mouse in the range 0 (fastest) to 100 (slowest). Note: A speed of 0 will move the mouse instantly. If omitted, the default speed (as set by SetDefaultMouseSpeed or 2 otherwise) will be used. Speed is ignored for SendInput/Play modes; they move the mouse instantaneously (though SetMouseDelay has a mode that applies to SendPlay). To visually move the mouse more slowly -- such as a script that performs a demonstration for an audience -- use SendEvent "{Click 100 200}" or SendMode "Event" (optionally in conjuction with BlockInput). DownOrUp Type: String If omitted, each click will consist of a "down" event followed by an "up" event. To change this behavior, specify one of the following letters: D: Press the mouse button down but do not release it (i.e. generate a down-event). U: Release the mouse button (i.e. generate an up-event). Relative Type: String If omitted, the X and Y coordinates will be treated as absolute values. To change this behavior, specify the following letter: R: The X and Y coordinates will be treated as offsets from the current mouse position. In other words, the cursor will be moved from its current position by X pixels to the right (left if negative) and Y pixels down (up if negative). Remarks This function uses the sending method set by SendMode. The Click function is recommended over MouseClick because it is generally easier to use. However, MouseClick supports the Speed parameter, whereas adjusting the speed of movement by Click requires the use of SetDefaultMouseSpeed. To perform a shift-click or control-click, use the Send function before and after the operation as shown in these examples: ; Example #1: Send "{Control down}" MouseClick "left", 55, 233 Send "{Control up}" ; Example #2: Send "{Shift down}" MouseClick "left", 55, 233 Send "{Shift up}" The SendPlay mode is able to successfully generate mouse events in a broader variety of games than the other modes. In addition, some applications and games may have trouble tracking the mouse if it moves too quickly. The speed parameter or SetDefaultMouseSpeed can be used to reduce the speed (in the default SendEvent mode only). Some applications do not obey a ClickCount higher than 1 for the mouse wheel. For them, use a Loop such as the following: Loop 5 MouseClick "WheelUp" The BlockInput function can be used to prevent any physical mouse activity by the user from disrupting the simulated mouse events produced by the mouse functions. However, this is generally not needed for the SendInput/Play modes because they automatically postpone the user's physical mouse activity until afterward. There is an automatic delay after every click-down and click-up of the mouse (except for SendInput mode and for turning the mouse wheel). Use SetMouseDelay to change the length of the delay. Related CoordMode, SendMode, SetDefaultMouseSpeed, SetMouseDelay, Click, MouseClickDrag, MouseGetPos, MouseMove, ControlClick, BlockInput Examples Double-clicks at the current mouse position. MouseClick "left" MouseClick "left" Same as above. MouseClick "left",,, 2 Moves the mouse cursor to a specific position, then right-clicks once. MouseClick "right", 200, 300 Simulates the turning of the mouse wheel. #up::MouseClick "WheelUp",,, 2 ; Turn it by two notches. #down::MouseClick "WheelDown",,, 2 MouseClickDrag - Syntax & Usage | AutoHotkey v2 MouseClickDrag Clicks and holds the specified mouse button, moves the mouse to the destination coordinates, then releases the button. MouseClickDrag WhichButton, X1, Y1, X2, Y2 , Speed, Relative Parameters WhichButton Type: String The button to click: Left, Right, Middle (or just the first letter of each of these). Specify X1 for the fourth button and X2 for the fifth. For example: MouseClickDrag "X1", ... Left and Right correspond to the primary button and secondary button. If the user swaps the buttons via system settings, the physical positions of the buttons are swapped but the effect stays the same. X1, Y1 Type: Integer The x/y coordinates of the drag's starting position (the mouse will be moved to these coordinates right before the drag is started). Coordinates are relative to the active window's client area unless CoordMode was used to change that. If omitted, the mouse's current position is used. X2, Y2 Type: Integer The x/y coordinates to drag the mouse to (that is, while the button is held down). Coordinates are relative to the active window's client area unless CoordMode was used to change that. Speed Type: Integer The speed to move the mouse in the range 0 (fastest) to 100 (slowest). Note: A speed of 0 will move the mouse instantly. If omitted, the default speed (as set by SetDefaultMouseSpeed or 2 otherwise) will be used. Speed is ignored for SendInput/Play modes; they move the mouse instantaneously (though SetMouseDelay has a mode that applies to SendPlay). To visually move the mouse more slowly -- such as a script that performs a demonstration for an audience -- use SendEvent {Click 100 200} or SendMode Event (optionally in conjuction with BlockInput). Relative Type: String If omitted, the X and Y coordinates will be treated as absolute values. To change this behavior, specify the following letter: R: The X1 and Y1 coordinates will be treated as offsets from the current mouse position. In other words, the cursor will be moved from its current position by X1 pixels to the right (left if negative) and Y1 pixels down (up if negative). Similarly, the X2 and Y2 coordinates will be treated as offsets from the X1 and Y1 coordinates. For example, the following would first move the cursor down and to the right by 5 pixels from its starting position, and then drag it from that position down and to the right by 10 pixels: MouseClickDrag "Left", 5, 5, 10, 10, , "R". Remarks This function uses the sending method set by SendMode. Dragging can also be done via the various Send functions, which is more flexible because the mode can be specified via the function name. For example: SendEvent "{Click 6 52 Down}{click 45 52 Up}" Another advantage of the method above is that unlike MouseClickDrag, it automatically compensates when the user has swapped the left and right mouse buttons via the system's control panel. The SendPlay mode is able to successfully generate mouse events in a broader variety of games than the other modes. However, dragging via SendPlay might not work in RichEdit

controls (and possibly others) such as those of WordPad and Metapad. Some applications and games may have trouble tracking the mouse if it moves too quickly. The speed parameter or SetDefaultMouseSpeed can be used to reduce the speed (in the default SendEvent mode only). The BlockInput function can be used to prevent any physical mouse activity by the user from disrupting the simulated mouse events produced by the mouse functions. However, this is generally not needed for the SendInput/Play modes because they automatically postpone the user's physical mouse activity until afterward. There is an automatic delay after every click-down and click-up of the mouse (except for SendInput mode). This delay also occurs after the movement of the mouse during the drag operation. Use SetMouseDelay to change the length of the delay. Related CoordMode, SendMode, SetDefaultMouseSpeed, SetMouseDelay, Click, MouseClick, MouseGetPos, MouseMove, BlockInput Examples Clicks and holds the left mouse button, moves the mouse cursor to the destination coordinates, then releases the button. MouseClickDrag "left", 0, 200, 600, 400 Opens MS Paint and draws a little house. Run "mspaint.exe" if !WinWaitActive("ahk_class MSPaintApp"„ 2) return WinMaximize MouseClickDrag "L", 150, 350, 150, 250 MouseClickDrag "L", 150, 250, 200, 200 MouseClickDrag "L", 200, 200, 250, 250 MouseClickDrag "L", 250, 250, 150, 250 MouseClickDrag "L", 150, 250, 250, 350 MouseClickDrag "L", 250, 350, 250, 250 MouseClickDrag "L", 250, 250, 150, 350 MouseClickDrag "L", 150, 350, 250, 350 MouseGetPos - Syntax & Usage | AutoHotkey v2 MouseGetPos Retrieves the current position of the mouse cursor, and optionally which window and control it is hovering over. MouseGetPos &OutputVarX, &OutputVarY, &OutputVarWin, &OutputVarControl, Flag Parameters &OutputVarX, &OutputVarY Type: VarRef References to the output variables in which to store the X and Y coordinates. The retrieved coordinates are relative to the active window's client area unless CoordMode was used to change to screen coordinates. &OutputVarWin Type: VarRef This optional parameter is a reference to the output variable in which to store the unique ID number of the window under the mouse cursor. If the window cannot be determined, this variable will be made blank. The window does not have to be active to be detected. Hidden windows cannot be detected. &OutputVarControl Type: VarRef This optional parameter is a reference to the output variable in which to store the name (ClassNN) of the control under the mouse cursor. If the control cannot be determined, this variable will be made blank. The names of controls should always match those shown by the Window Spy. The window under the mouse cursor does not have to be active for a control to be detected. Flag Type: Integer If omitted or 0, the function uses the default method to determine OutputVarControl and stores the control's ClassNN. To change this behavior, add up one or both of the following digits: 1: Uses a simpler method to determine OutputVarControl. This method correctly retrieves the active/topmost child window of an Multiple Document Interface (MDI) application such as SysEdit or TextPad. However, it is less accurate for other purposes such as detecting controls inside a GroupBox control. 2: Stores the control's HWND in OutputVarControl rather than the control's ClassNN. For example, to put both options into effect, the Flag parameter must be set to 3. Remarks Any of the output variables may be omitted if the corresponding information is not needed. On systems with multiple screens which have different DPI settings, the returned position may be different than expected due to OS DPI scaling. Related CoordMode, Win functions, SetDefaultMouseSpeed, Click Examples Reports the position of the mouse cursor. MouseGetPos &xpos, &ypos MsgBox "The cursor is at X" xpos " Y" ypos Allows you to move the mouse cursor around to see the title of the window currently under the cursor. SetTimer WatchCursor, 100 WatchCursor() { MouseGetPos , , &id, &control ToolTip ("ahk_id " id " ahk_class " WinGetClass(id) " " WinGetTitle(id) " Control: " control) } MouseMove - Syntax & Usage | AutoHotkey v2 MouseMove Moves the mouse cursor. MouseMove X, Y , Speed, Relative Parameters X, Y Type: Integer The x/y coordinates to move the mouse to. Coordinates are relative to the active window's client area unless CoordMode was used to change that. Speed Type: Integer The speed to move the mouse in the range 0 (fastest) to 100 (slowest). Note: A speed of 0 will move the mouse instantly. If omitted, the default speed (as set by SetDefaultMouseSpeed or 2 otherwise) will be used. Speed is ignored for SendInput/Play modes; they move the mouse instantaneously (though SetMouseDelay has a mode that applies to SendPlay). To visually move the mouse more slowly -- such as a script that performs a demonstration for an audience -- use SendEvent {Click 100 200} or SendMode Event (optionally in conjuction with BlockInput). Relative Type: String If omitted, the X and Y coordinates will be treated as absolute values. To change this behavior, specify the following letter: R: The X and Y coordinates will be treated as offsets from the current mouse position. In other words, the cursor will be moved from its current position by X pixels to the right (left if negative) and Y pixels down (up if negative). Remarks This function uses the sending method set by SendMode. The SendPlay mode is able to successfully generate mouse events in a broader variety of games than the other modes. In addition, some applications and games may have trouble tracking the mouse if it moves too quickly. The speed parameter or SetDefaultMouseSpeed can be used to reduce the speed (in the default SendEvent mode only). The BlockInput function can be used to prevent any physical mouse activity by the user from disrupting the simulated mouse events produced by the mouse functions. However, this is generally not needed for the SendInput/Play modes because they automatically postpone the user's physical mouse activity until afterward. There is an automatic delay after every movement of the mouse (except for SendInput mode). Use SetMouseDelay to change the length of the delay. The following is an alternate way to move the mouse cursor that may work better in certain multi-monitor configurations: DllCall("SetCursorPos", "int", 100, "int", 400) ; The first number is the X-coordinate and the second is the Y (relative to the screen). On a related note, the mouse cursor can be temporarily hidden via the hide-cursor example. Related CoordMode, SendMode, SetDefaultMouseSpeed, SetMouseDelay, Click, MouseClick, MouseClickDrag, MouseGetPos, BlockInput Examples Moves the mouse cursor to a new position. MouseMove 200, 100 Moves the mouse cursor slowly (speed 50 vs. 2) by 20 pixels to the right and 30 pixels down from its current location. MouseMove 20, 30, 50, "R" MsgBox - Syntax & Usage | AutoHotkey v2 MsgBox Displays the specified text in a small window containing one or more buttons (such as Yes and No). MsgBox Text, Title, Options Result := MsgBox(Text, Title, Options) Parameters Text Type: String The text to display inside the message box. If omitted, the default text is "Press OK to continue." if "OK" is the only button present, or blank otherwise. Escape sequences can be used to denote special characters. For example, `n indicates a linefeed character, which ends the current line and begins a new one. Thus, using text1`n`ntext2 would create a blank line between text1 and text2. If Text is long, it can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. Title Type: String The title of the message box window. If omitted, it defaults to the current value of A_ScriptName. Options Type: String Indicates the type of message box and the possible button combinations. If blank or omitted, it defaults to 0. See the tables below for allowed values. In addition, zero or more of the following options can be specified: Owner: To specify an owner window for the message box, use the word Owner followed immediately by a HWND (window ID). T: Timeout. To have the message box close automatically if the user has not closed it within a specified time, use the letter T followed by the timeout in seconds, which can contain a decimal point. If this value exceeds 2147483 (24.8 days), it will be set to 2147483. If the message box times out, the return value is the word Timeout. Values for the Options parameter The Options parameter can be either a combination (sum) of numeric values from the following groups, which is passed directly to the operating system's MessageBox function, or a string of zero or more case-insensitive options separated by at least one space or tab. One or more numeric options may also be included in the string. Group #1: Buttons To indicate the buttons displayed in the message box, add one of the following values: Function Dec Hex String OK (that is, only an OK button is displayed) 0 0x0 OK or O OK, Cancel 1 0x1 OKCancel, O/C or OC Abort, Retry, Ignore 2 0x2 AbortRetryIgnore, A/R/I or ARI Yes, No, Cancel 3 0x3 YesNoCancel, Y/N/C or YNC Yes, No 4 0x4 YesNo, Y/N or YN Retry, Cancel 5 0x5 RetryCancel, R/C or RC Cancel, Try Again, Continue 6 0x6 CancelTryAgainContinue, C/T/C or CTC Group #2: Icon To display an icon in the message box, add one of the following values: Function Dec Hex String Icon Hand (stop/error) 16 0x10 Iconx Icon Question 32 0x20 Icon? Icon Exclamation 48 0x30 Icon! Icon Asterisk (info) 64 0x40 Iconi Group #3: Default Button To indicate the default button, add one of the following values: Function Dec Hex String Makes the 2nd button the default 256 0x100 Default2 Makes the 3rd button the default 512 0x200 Default3 Makes the 4th button the default(requires the Help button to be present) 768 0x300 Default4 Group #4: Modality To indicate the modality of the dialog box, add one of the following values: Function Dec Hex String System Modal (always on top) 4096 0x1000 N/A Task Modal 8192 0x2000 N/A Always-on-top (style WS_EX_TOPMOST)(like System Modal but omits title bar icon) 262144 0x40000 N/A Group #5: Other Options To specify other options, add one or more of the following values: Function Dec Hex String Adds a Help button (see remarks below) 16384 0x4000 N/A Makes the text right-justified 524288 0x80000 N/A Right-to-left reading order for Hebrew/Arabic 1048576 0x100000 N/A Return Value Type: String This function returns one of the following strings to represent which button the user pressed: OK Cancel Yes No Abort Retry Ignore TryAgain Continue Timeout (that is, the word "timeout" is returned if the message box timed out) If the dialog could not be displayed, an empty string is returned. This typically only occurs as a result of the MsgBox limit being reached, but may occur in other unusual cases. Error Handling An Error is thrown on failure, such as if the options are invalid, the MsgBox limit has been reached, or the message box could not be displayed for some other reason. Remarks A message box usually looks like this: To determine which button the user pressed, use the function's return value. For example: Result := MsgBox("Would you like to continue? (press Yes or No)"„ "YesNo") if Result = "Yes" MsgBox "You pressed Yes." else MsgBox "You pressed No." if MsgBox("Retry or cancel?"„ "R/C") = "Retry" MsgBox("You pressed Retry.") The names of the buttons can be customized by following this example. Tip: Pressing Ctrl+C while a message box is active will copy its text to the clipboard. This applies to all message boxes, not just those produced by AutoHotkey. Using MsgBox with GUI windows: A GUI window may display a modal message box by means of the OwnDialogs option. A modal message box prevents the user from interacting with the GUI window until the message box is dismissed. In such a case, it is not necessary to specify the System Modal or Task Modal options from the table above. When the OwnDialogs option is not in effect, the Task Modal option (8192) can be used to disable all the script's windows until the user dismisses the message box. If the OwnerHWND option is specified, it takes precedence over any other setting. HWND can be the HWND of any window, even one not owned by the script. The Help button: When the Help button option (16384) is present in Options, pressing the Help button will have no effect unless both of the following are true: The message box is owned by a GUI window by means of the OwnDialogs option. The script is monitoring the WM_HELP message (0x0053). For example: OnMessage(0x0053, "WM_HELP"). When the WM_HELP function is called, it may guide the user by means such as showing another window or message box. The Close button (in the message box's title bar): Since the message box is a built-in feature of the operating system, its X button is enabled only when certain buttons are present. If there is only an OK button, clicking the X button is the same as pressing OK. Otherwise, the X button is disabled unless there is a Cancel button, in which case clicking the X is the same as pressing Cancel. Maximum 7 ongoing calls: The thread displaying a message box can typically be interrupted, allowing the new thread to display its own message box before the previous call returns. A

maximum of 7 ongoing calls to MsgBox are permitted, but any calls beyond the 7th cause an Error to be thrown. Note that a call to MsgBox in an interrupted thread cannot return until the thread is resumed. Related InputBox, FileSelect, DirSelect, ToolTip, Gui object Examples Shows a message box with specific text. A quick and easy way to show information. The user can press an OK button to close the message box and continue execution. MsgBox "This is a string." Shows a message box with specific text and a title. MsgBox "This MsgBox has a custom title.", "A Custom Title" Shows a message box with default text. Mainly useful for debugging purposes, for example to quickly set a breakpoint in the script. MsgBox ; "Press OK to continue." Shows a message box with specific text, a title and an info icon. Besides, a continuation section is used to display the multi-line text in a more clear manner. MsgBox " (The first parameter is displayed as the message. The second parameter becomes the window title. The third parameter determines the type of message box.)", "Window Title", "iconi" Use the return value to determine which button the user pressed in the message box. Note that in this case the MsgBox function call must be specified with parentheses. result := MsgBox ("Do you want to continue? (Press YES or NO)",, "YesNo") if (result = "No") return Use the T (timeout) option to automatically close the message box after a certain number of seconds. result := MsgBox("This MsgBox will time out in 5 seconds. Continue?",, "Y/N T5") if (result = "Timeout") MsgBox "You didn't press YES or NO within the 5-second period." else if (result = "No") return Include a variable or sub-expression in the message. See also: Concatenation var := 10 MsgBox "The initial value is: " var MsgBox "The result is: " var * 2 MsgBox Format("The result is: {1}", var * 2) Number - Syntax & Usage | AutoHotkey v2 Number Converts a numeric string to a pure integer or floating-point number. NumValue := Number(Value) Return Value Type: Integer or Float This function returns the result of converting Value to a pure integer or floating-point number, or Value itself if it is already an Integer or Float value. Remarks If the value cannot be converted, a TypeError is thrown. To determine if a value can be converted to a number, use the IsNumber function. Number is actually a class, but can be called as a function. Value is Number can be used to check whether a value is a pure number. Related Type, Float, Integer, String, Values, Expressions, Is functions NumGet - Syntax & Usage | AutoHotkey v2 NumGet Returns the binary number stored at the specified address+offset. Number := NumGet(Source, Offset, Type) Parameters Source Type: Object or Integer A Buffer-like object or memory address. Any object which implements Ptr and Size properties may be used, but this function is optimized for the native Buffer object. Passing an object with these properties ensures that the function does not read memory from an invalid location; doing so could cause crashes or other unpredictable behaviour. Offset Type: Integer An offset - in bytes - which is added to Source to determine the source address. If omitted, it defaults to 0. Type Type: String One of the following strings: UInt, Int, Int64, Short, UShort, Char, UChar, Double, Float, Ptr or UPtr Unsigned 64-bit integers are not supported, as AutoHotkey's native integer type is Int64. Therefore, to work with numbers greater than or equal to 0x8000000000000000, omit the U prefix and interpret any negative values as large integers. For example, a value of -1 as an Int64 is really 0xFFFFFFFFFFFFFFFF if it is intended to be a UInt64. On 64-bit builds, UPtr is equivalent to Int64. These type names must be enclosed in quotes when used as literal strings. For details see DllCall Types. Return Value Type: Integer or Float This function returns the binary number at the specified address+offset. General Remarks If only two parameters are present, the second parameter must be Type. For example, NumGet(var, "int") is valid. An exception may be thrown if the source address is invalid. However, some invalid addresses cannot be detected as such and may cause unpredictable behaviour. Passing a Buffer object instead of an address ensures that the source address can always be validated. Related NumPut, DllCall, Buffer object, VarSetStrCapacity NumPut - Syntax & Usage | AutoHotkey v2 NumPut Stores one or more numbers in binary format at the specified address+offset. NumPut Type, Number, Type2, Number2, ... Target , Offset Parameters Type Type: String One of the following strings: UInt, UInt64, Int, Int64, Short, UShort, Char, UChar, Double, Float, Ptr or UPtr For all integer types, or when passing pure integers, signed vs. unsigned does not affect the result due to the use of two's complement to represent signed integers. These type names must be enclosed in quotes when used as literal strings. For details see DllCall Types. Number Type: Integer The number to store. Target Type: Object or Integer A Buffer-like object or memory address. Any object which implements Ptr and Size properties may be used, but this function is optimized for the native Buffer object. Passing an object with these properties ensures that the function does not write to an invalid memory location; doing so could cause crashes or other unpredictable behaviour. Offset Type: Integer An offset - in bytes - which is added to Target to determine the target address. If omitted, it defaults to 0. Return Value Type: Integer This function returns the address to the right of the last item written. This can be used when writing a non-contiguous sequence of numbers, such as in a structure for use with DllCall, where some fields are not being set. However, in many cases it is simpler and more efficient to specify multiple Type, Number pairs instead. Passing the address back to NumPut is less safe than passing a Buffer-like object with an updated Offset. General Remarks A sequence of numbers can be written by repeating Type and Number any number of times after the first Number. Each number is written at the next byte after the previous number, with no padding. When creating a structure for use with DllCall, be aware that some fields may need explicit padding added due to data alignment requirements. If an integer is too large to fit in the specified Type, its most significant bytes are ignored; e.g. NumPut("Char", 257, buf) would store the number 1. An exception may be thrown if the target address is invalid. However, some invalid addresses cannot be detected as such and may cause unpredictable behaviour. Passing a Buffer object instead of an address ensures that the target address can always be validated. Related NumGet, DllCall, Buffer object, VarSetStrCapacity ObjAddRef / ObjRelease - Syntax & Usage | AutoHotkey v2 ObjAddRef / ObjRelease Increments or decrements an object's reference count. NewRefCount := ObjAddRef(Ptr) NewRefCount := ObjRelease(Ptr) Parameters Ptr Type: Integer An unmanaged object pointer or COM interface pointer. Return Value Type: Integer These functions return the new reference count. This value should be used only for debugging purposes. Related Reference Counting Although the following articles discuss reference counting as it applies to COM, they cover some important concepts and rules which generally also apply to AutoHotkey objects: IUnknown::AddRef, IUnknown::Release, Reference Counting Rules. ObjBindMethod - Syntax & Usage | AutoHotkey v2 ObjBindMethod Creates a BoundFunc object which calls a method of a given object. BoundFunc := ObjBindMethod(Obj , Method, Params) Parameters Obj Type: Object Any object. Method Type: String A method name. If omitted, the bound function calls Obj itself. Params Any number of parameters. Remarks For details and examples, see BoundFunc object. Object - Methods & Properties | AutoHotkey v2 Object class Object extends Any Object is the basic class from which other AutoHotkey object classes derive. Each instance of Object consists of a set of "own properties" and a base object, from which more properties are inherited. Objects also have methods, but these are just properties which can be called. There are value properties and dynamic properties. Value properties simply contain a value. Dynamic properties do not contain a value, but instead call an accessor function depending on how they are accessed (get, set or call). "Obj" is used below as a placeholder for any instance of the Object class. All instances of Object are based on Object.Prototype, which is based on Any.Prototype. In addition to the methods and property inherited from Any, Objects have the following predefined methods and properties. Table of Contents Static Methods: Call: Creates a new Object. Methods: Clone: Returns a shallow copy of an object. DefineProp: Defines a new own property. DeleteProp: Removes an own property from an object. GetOwnPropDesc: Returns a descriptor for a given own property, compatible with DefineProp. HasOwnProp: Returns true if an object owns a property by the specified name. OwnProps: Enumerates an object's own properties. Properties: Base: Retrieves or sets an object's base object. Functions: ObjSetBase: Set an object's base object. ObjGetCapacity, ObjSetCapacity: Retrieve or set an Object's capacity to contain properties. ObjOwnPropCount: Retrieve the number of own properties contained by an object. ObjHasOwnProp, ObjOwnProps: Equivalent to the corresponding predefined method, but cannot be overridden. Static Methods Call Creates a new Object. Obj := Object() Methods Clone Returns a shallow copy of an object. Clone := Obj.Clone() Each property or method owned by the object is copied into the clone. Object references are copied (like with a normal assignment), not the objects themselves; in other words, if a property contains a reference to an object, the clone will contain a reference to the same object. Dynamic properties are copied, not invoked. The clone has the same base object as the original object. DefineProp Defines a new own property. Obj.DefineProp(Name, Desc) Name Type: String The name of the property. Desc Type: Object An object with one of the following own properties, or both Get and Set: Get: The function object to call when the property's value is retrieved. Set: The function object to call when the property is assigned a value. Its second parameter is the value being assigned. Call: The function object to call when the property is called. Value: Any value to assign to the property. Returns the target object (Obj). This method can be used to convert a value property to a dynamic property or vice versa, but it is not possible to specify both a value and accessor functions. If any one of the accessor functions is omitted, behavior is inherited from a base object. An inherited value property is equivalent to a set of accessor functions which return or call the value, or store a new value in this. Note that a new value would overwrite any dynamic property in this itself, and override any inherited accessor functions. If no Set or value is defined or inherited, attempting to set the property will throw an exception. If no Call is defined or inherited, Get may be called to retrieve a function object, which is then called. If no Get is defined or inherited but there is a Call accessor function, the function itself becomes the property's value (read-only). As with methods, the first parameter of Get, Set or Call is this (the target object). For Set, the second parameter is value (the value being assigned). These parameters are defined automatically by method and property definitions within a class, but must be defined explicitly if using normal functions. Any other parameters passed by the caller are appended to the parameter list. The MaxParams and IsVariadic properties of the function objects are evaluated to determine whether the property may accept parameters. If MaxParams is 1 for get or 2 for set and IsVariadic is false or undefined, the property cannot accept parameters; they are instead forwarded to the __Item property of the object returned by get. DeleteProp Removes an own property from an object. Obj.DeleteProp(Name) Name Type: String A property name. Returns the value of the removed property (blank if none). GetOwnPropDesc Returns a descriptor for a given own property, compatible with DefineProp. Obj.GetOwnPropDesc(Name) Name Type: String A property name. For a dynamic property, the return value is a new object with an own property for each accessor function: Get, Set, Call. Each property is present only if the corresponding accessor function is defined in Obj itself. For a value property, the return value is a new object with a property named Value. In such cases, Obj.GetOwnPropDesc(Name).Value == Obj.%Name%. Modifying the returned object has no effect on Obj unless DefineProp is called. A PropertyError is thrown if Obj does not own a property by that name. The script can determine whether a property is dynamic by checking not desc.HasProp("Value"), where desc is the return value of GetOwnPropDesc. HasOwnProp Returns true if an object owns a property by the specified name, otherwise false. Obj.HasOwnProp(Name) The default implementation of this method is also defined as a function: ObjHasOwnProp(Obj, Name). OwnProps Enumerates an object's own properties. For Name , Value in Obj.OwnProps() Returns a new enumerator. The enumerator is typically passed directly to

a for-loop, which calls the enumerator once for each iteration of the loop. Each call to the enumerator returns the next property name and/or value. The for-loop's variables correspond to the enumerator's parameters, which are: Name Type: String The property's name. Value The property's value. If the property has a getter method, it is called to obtain the value (unless Value is omitted). Dynamic properties are included in the enumeration. However: Since only the object's own properties are enumerated, the property must be defined directly in Obj. If only the first variable was specified, the property's name is returned and its getter is not called. If two variables were specified, the enumerator attempts to call the property's getter to retrieve the value. If the getter requires parameters, the property is skipped. If Obj itself does not define a getter for this property, it is skipped. Note: Properties defined by a method definition typically do not have a getter, so are skipped. If Obj is a class prototype object, the getter should not (and in some cases cannot) be called; so the property is skipped. If the getter throws an exception, it is propagated (not suppressed). The caller can continue enumeration at the next property only if it retained a reference to the enumerator (i.e. not if it passed the enumerator directly to a for-loop, since in that case the enumerator is freed when the for-loop aborts). To enumerate own properties without calling property getters, or to enumerate all properties regardless of type, pass only a single variable to the for-loop or enumerator. GetOwnPropDesc can be used to differentiate value properties from dynamic properties, while also retrieving the value or getter/setter/method. Methods are typically excluded from enumeration in the two-parameter mode, because evaluation of the property normally depends on whether the object has a corresponding getter or value, either in the same object or a base object. To avoid inconsistency when two variables are specified, OwnProps skips over own properties that have only a Call accessor function. For example: If OwnProps returned the method itself when no getter is defined, defining a getter would then prevent the method from being returned. Scripts relying on the two variable mode to retrieve methods would then miss some methods. If OwnProps returned the method itself when a getter is defined by a base object, this would be inconsistent with normal evaluation of the property. The default implementation of this method is also defined as a function: ObjOwnProps(Obj). Properties Base Retrieves or sets an object's base object. BaseObj := Obj.Base Obj.Base := BaseObj BaseObj must be an Object. If assigning the new base would change the native type of the object, an exception is thrown. An object's native type is decided by the nearest prototype object belonging to a built-in class, such as Object.Prototype or Array.Prototype. For example, an instance of Array must always derive from Array.Prototype, either directly or indirectly. Properties and methods are inherited from the base object dynamically, so changing an object's base also changes which inherited properties and methods are available. This property is inherited from Any; however, it can be set only for instances of Object. See also: ObjGetBase, ObjSetBase Functions ObjSetBase Sets an object's base object. ObjSetBase(Obj, BaseObj) No meta-functions or property functions are called. Overriding the Base property does not affect the behaviour of this function. An exception is thrown if Obj or BaseObj is of an incorrect type. See also: ObjGetBase, Base property ObjOwnPropCount Returns the number of properties owned by an object. Count := ObjOwnPropCount(Obj) ObjSetCapacity Sets the current capacity of the object's internal array of own properties. ObjSetCapacity(Obj, MaxProps) MaxProps Type: Integer The new capacity. If less than the current number of own properties, that number is used instead, and any unused space is freed. Returns the new capacity. An exception is thrown if Obj is of an incorrect type. ObjGetCapacity MaxItems := ObjGetCapacity(Obj) Returns the current capacity of the object's internal array of properties. An exception is thrown if Obj is of an incorrect type. OnClipboardChange - Syntax & Usage | AutoHotkey v2 OnClipboardChange Causes the specified function to be called automatically whenever the clipboard's content changes. OnClipboardChange Function , AddRemove Parameters Function Type: Function Object The function object to call. The function's parameter and return value are described below. AddRemove Type: Integer If omitted, it defaults to 1 (call the function after any previously registered functions). Otherwise, specify one of the following numbers: 1 = Call the function after any previously registered functions. -1 = Call the function before any previously registered functions. 0 = Do not call the function. Function The function should accept one parameter: FunctionName(Type) Type Type: Integer One of the following numbers: 0 = Clipboard is now empty. 1 = Clipboard contains something that can be expressed as text (this includes files copied from an Explorer window). 2 = Clipboard contains something entirely non-text such as a picture. Return Value If this is the last or only OnClipboardChange function, the return value is ignored. Otherwise, the function can return a non-zero integer to prevent subsequent functions from being called. Remarks If the clipboard changes while an OnClipboardChange function is already running, that notification event is lost. If this is undesirable, use Critical. However, this will also buffer/defer other threads (such as the press of a hotkey) that occur while the OnClipboardChange thread is running. If the script itself changes the clipboard, its OnClipboardChange functions are typically not executed immediately; that is, functions immediately below the function that changed the clipboard are likely to execute beforehand. To force the functions to execute immediately, use a short delay such as Sleep 20 after changing the clipboard. Related A_Clipboard, OnExit, OnMessage, CallbackCreate Examples Briefly displays a tooltip for each clipboard change. OnClipboardChange ClipChanged ClipChanged(clip_type) { ToolTip "Clipboard data type: " clip_type Sleep 1000 ToolTip ; Turn off the tip. } OnError - Syntax & Usage | AutoHotkey v2 OnError Causes the specified function to be called automatically when an unhandled error occurs. OnError Function , AddRemove Parameters Function Type: Function Object The function object to call when an unhandled error occurs. The function's parameters and return value are described below. AddRemove Type: Integer If omitted, it defaults to 1 (call the function after any previously registered functions). Otherwise, specify one of the following numbers: 1 = Call the function after any previously registered functions. -1 = Call the function before any previously registered functions. 0 = Do not call the function. Function The function should accept two parameters: FunctionName(Thrown, Mode) Thrown Type: Any The thrown value, usually an Error object. Mode Type: String One of the following strings: "Return": Thrown is a continuable runtime error. The thread continues if the callback returns -1; otherwise the thread exits. "Exit": Thrown is a non-continuable runtime error or a value thrown by the script. The thread will exit. "ExitApp": Thrown is a critical runtime error, such as corruption detected by DllCall. The program will exit. Return Value The callback function can return one of the following values (other values are reserved for future use and should be avoided): 0, "" or no return: Allow error handling to proceed as normal. 1: Suppress the default error dialog and any remaining error callbacks. -1: As above, but if Mode is "Return", execution of the current thread is permitted to continue. Remarks Function is called only for errors or exceptions which would normally cause an error message to be displayed. It cannot be called for a load-time error, since OnError cannot be called until after the script has loaded. Function is called on the current thread, before it exits (that is, before the call stack unwinds). Related Try, Catch, Throw, OnExit Examples Logs errors caused by the script into a text file instead of displaying them to the user. OnError LogError i := Integer("cause_error") LogError (exception, mode) { FileAppend "Error on line " exception.Line ": " exception.Message "``n" , "errorlog.txt" return true } Use OnError to implement alternative error handling methods. Caveat: OnError is ineffective while Try is active. AccumulateErrors() { local ea := ErrorAccumulator() ea.Start() return ea } class ErrorAccumulator { Errors := [] ; Array for accumulated errors. _cb := AccumulateError.Bind(this.Errors) Start() => OnError(this._cb, -1) ; Register our cb before others. Stop() => OnError(this._cb, 0) ; Unregister our cb. Last => this.Errors[-1] ; Most recent error. Count => this.Errors.Length ; Number of accumulated errors. __item[i] => this.Errors[i] ; Shortcut for indexing. __delete() => this.Stop() ; For tying to function scope. } ; This is the OnError callback function. 'errors' is given a value via Bind(). AccumulateError(errors, e, mode) { if mode != "Return" ; Not continuable. return if e.What = "" ; Expression defect or similar, not a built-in function. return try { ; Try to print the error to stdout. FileAppend Format("{1} ({2}) : ({3}) {4}``n", e.File, e.Line, e.What, e.Message), "*" if HasProp(e, "extra") FileAppend " Specifically: " e.Extra "``n", "*" } errors.Push(e) return -1 ; Continue. } RearrangeWindows() { ; Start accumulating errors in 'err'. local err := AccumulateErrors() ; Do some things that might fail... MonitorGetWorkArea , &left, &top, &right, &bottom width := (right-left)//2, height := bottom-top WinMove left, top, width, height, A_ScriptFullPath WinMove left+width, top, width, height, "AutoHotkey v2 Help" ; Check if any errors occurred. if err.Count MsgBox err.Count " error(s); last error at line #" err.Last.Line else MsgBox "No errors" ; Stop is called automatically when the variable goes out of scope, ; since only we have a reference to the object. This causes OnError ; to be called to unregister the callback. ;err.Stop() } ; Call the test function which suppresses and accumulates errors. RearrangeWindows() ; Call another function to show normal error behaviour is restored. WinMove 0, 0, 0, 0, "non-existent window" OnExit - Syntax & Usage | AutoHotkey v2 OnExit Causes the specified function to be called automatically when the script exits. OnExit Function , AddRemove Parameters Function Type: Function Object The function object to call when the script is exiting. The function's parameters and return value are described below. AddRemove Type: Integer If omitted, it defaults to 1 (call the function after any previously registered functions). Otherwise, specify one of the following numbers: 1 = Call the function after any previously registered functions. -1 = Call the function before any previously registered functions. 0 = Do not call the function. Function The function should accept two parameters: FunctionName(ExitReason, ExitCode) ExitReason Type: String One of the following words: Word Meaning Logoff The user is logging off. Shutdown The system is being shut down or restarted, such as by the Shutdown function. Close The script was sent a WM_CLOSE or WM_QUIT message, had a critical error, or is being closed in some other way. Although all of these are unusual, WM_CLOSE might be caused by WinClose having been used on the script's main window. To close (hide) the window without terminating the script, use WinHide. If the script is exiting due to a critical error or its main window being destroyed, it will unconditionally terminate after the OnExit thread completes. If the main window is being destroyed, it may still exist but cannot be displayed. This condition can be detected by monitoring the WM_DESTROY message with OnMessage. Error A runtime error occurred in a script that is not persistent. An example of a runtime error is Run/RunWait being unable to launch the specified program or document. Menu The user selected Exit from the main window's menu or from the standard tray menu. Exit The Exit or ExitApp function was used (includes custom menu items). Reload The script is being reloaded via the Reload function or menu item. Single The script is being replaced by a new instance of itself as a result of #SingleInstance. ExitCode Type: Integer An integer between -2147483648 and 2147483647 that is returned to its caller when the script exits. This code is accessible to any program that spawned the script, such as another script (via RunWait) or a batch (.bat) file. Zero is traditionally used to indicate success. Return Value The function can return a non-zero integer to prevent the script from exiting (with some rare exceptions) and subsequent functions from being called. Otherwise, the script exits after all registered functions are called. Remarks Any number of OnExit functions can be registered. An OnExit function usually should not call ExitApp; if it does, the script terminates immediately. OnExit functions are called when the script exits by any means (except when it is killed by something like "End Task"). It is also called whenever #SingleInstance and Reload ask a previous instance to terminate. A script can detect and optionally abort a system shutdown or logoff via OnMessage

(0x0011, On_WM_QUERYENDSESSION) (see OnMessage example #2 for a working script). The OnExit thread does not obey #MaxThreads (it will always launch when needed). In addition, while it is running, it cannot be interrupted by any thread, including hotkeys, custom menu items, and timed subroutines. However, it will be interrupted (and the script will terminate) if the user chooses Exit from the tray menu or main menu, or the script is asked to terminate as a result of Reload or #SingleInstance. Because of this, an OnExit function should be designed to finish quickly unless the user is aware of what it is doing. If the OnExit thread encounters a failure condition such as a runtime error, the script will terminate. If the OnExit thread was launched due to an Exit or ExitApp function that specified an exit code, that exit code is used unless an OnExit function returns true (preventing exit) or calls ExitApp. Whenever an exit attempt is made, each OnExit function starts off fresh with the default values for settings such as SendMode. These defaults can be changed during script startup. Related OnError, OnMessage, CallbackCreate, OnClipboardChange, ExitApp, Shutdown, Persistent, Threads, Return Examples Asks the user before exiting the script. To test this example, right-click the tray icon and click Exit. Persistent ; Prevent the script from exiting automatically. OnExit ExitFunc ExitFunc(ExitReason, ExitCode) { if ExitReason != "Logoff" and ExitReason != "Shutdown" { Result := MsgBox("Are you sure you want to exit?",, 4) if Result = "No" return 1 ; OnExit functions must return non-zero to prevent exit. } ; Do not call ExitApp -- that would prevent other OnExit functions from being called. } Registers a method to be called on exit. Persistent ; Prevent the script from exiting automatically. OnExit MyObject.Exiting class MyObject { static Exiting(*) { MsgBox "MyObject is cleaning up prior to exiting..." /* this.SayGoodbye() this.CloseNetworkConnections() */ } } OnMessage - Syntax & Usage | AutoHotkey v2 OnMessage Causes the specified function to be called automatically whenever the script receives the specified message. OnMessage MsgNumber, Function , MaxThreads Parameters MsgNumber Type: Integer The number of the message to monitor or query, which should be between 0 and 4294967295 (0xFFFFFFFF). If you do not wish to monitor a system message (that is, one below 0x0400), it is best to choose a number greater than 4096 (0x1000) to the extent you have a choice. This reduces the chance of interfering with messages used internally by current and future versions of AutoHotkey. Function Type: Function Object The function object to call when the message is received. The function's parameters and return value are described below. MaxThreads Type: Integer This integer is normally omitted, in which case the monitor function is limited to one thread at a time. This is usually best because otherwise, the script would process messages out of chronological order whenever the monitor function interrupts itself. Therefore, as an alternative to MaxThreads, consider using Critical as described below. If the monitor function directly or indirectly causes the message to be sent again while the function is still running, it is necessary to specify a MaxThreads value greater than 1 or less than -1 to allow the monitor function to be called for the new message (if desired). Messages sent (not posted) by the script's own process to itself cannot be delayed or buffered. Specify 0 to unregister the previously registered function identified by Function. By default, when multiple functions are registered for a single MsgNumber, they are called in the order that they were registered. To register a function to be called before any previously registered functions, specify a negative value for MaxThreads. For example, OnMessage Msg, Fn, -2 registers Fn to be called before any other functions previously registered for Msg, and allows Fn a maximum of 2 threads. However, if the function is already registered, the order will not change unless it is unregistered and then re-registered. Usage Any number of functions or function objects can monitor a given MsgNumber. Either of these two lines registers a function object to be called after any previously registered functions: OnMessage MsgNumber, Function ; Option 1 - omit MaxThreads OnMessage MsgNumber, Function, 1 ; Option 2 - specify MaxThreads 1 This registers a function object to be called before any previously registered functions: OnMessage MsgNumber, Function, -1 To unregister a function object, specify 0 for MaxThreads: OnMessage MsgNumber, Function, 0 The Function's Parameters A function assigned to monitor one or more messages should accept four parameters: FunctionName(wParam, lParam, msg, hwnd) Although the names you give the parameters do not matter, the following information is sequentially assigned to them: Parameter #1: The message's WPARAM value. Parameter #2: The message's LPARAM value. Parameter #3: The message number, which is useful in cases where a function monitors more than one message. Parameter #4: The HWND (unique ID) of the window or control to which the message was sent. The HWND can be used directly in a WinTitle parameter. WPARAM and LPARAM are unsigned 32-bit integers (from 0 to 232-1) or signed 64-bit integers (from -263 to 263-1) depending on whether the exe running the script is 32-bit or 64-bit. For 32-bit scripts, if an incoming parameter is intended to be a signed integer, any negative numbers can be revealed by following this example: if (A_PtrSize = 4 && wParam > 0x7FFFFFFF) ; Checking A_PtrSize ensures the script is 32-bit. wParam := -(~wParam) - 1 You can omit one or more parameters from the end of the list if the corresponding information is not needed, but in this case an asterisk must be specified as the final parameter. For example, a function defined as MyMsgMonitor(wParam, lParam, *) would receive only the first two parameters, and one defined as MyMsgMonitor(*) would receive none of them. Additional Information Available to the Function In addition to the parameters received above, the function may also consult the built-in variable A_EventInfo, which contains 0 if the message was sent via SendMessage. If sent via PostMessage, it contains the tick-count time the message was posted. A monitor function's last found window starts off as the parent window to which the message was sent (even if it was sent to a control). If the window is hidden but not a GUI window (such as the script's main window), turn on DetectHiddenWindows before using it. For example: DetectHiddenWindows True MsgParentWindow := WinExist() ; This stores the unique ID of the window to which the message was sent. What the Function Should Return If a monitor function uses Return without any parameters, or it specifies a blank value such as "" (or it never uses Return at all), the incoming message goes on to be processed normally when the function finishes. The same thing happens if the function Exits or causes a runtime error such as running a nonexistent file. By contrast, returning an integer causes it to be sent immediately as a reply; that is, the program does not process the message any further. For example, a function monitoring WM_LBUTTONDOWN (0x0201) may return an integer to prevent the target window from being notified that a mouse click has occurred. In many cases (such as a message arriving via PostMessage), it does not matter which integer is returned; but if in doubt, 0 is usually safest. The range of valid return values depends on whether the exe running the script is 32-bit or 64-bit. Non-empty return values must be between -231 and 232-1 for 32-bit scripts (A_PtrSize = 4) and between -263 and 263-1 for 64-bit scripts (A_PtrSize = 8). If there are multiple functions monitoring a given message number, they are called one by one until one returns a non-empty value. General Remarks Unlike a normal function-call, the arrival of a monitored message calls the function as a new thread. Because of this, the function starts off fresh with the default values for settings such as SendMode and DetectHiddenWindows. These defaults can be changed during script startup. Messages sent to a control (rather than being posted) are not monitored because the system routes them directly to the control behind the scenes. This is seldom an issue for system-generated messages because most of them are posted. If the script is intended to stay running in an idle state to monitor for incoming messages, it may be necessary to call the Persistent function to prevent the script from exiting. OnMessage does not automatically make the script persistent, as it is sometimes unnecessary or undesired. For instance, when OnMessage is used to monitor input to a GUI window (such as in the WM_LBUTTONDOWN example), it is often more appropriate to allow the script to exit automatically when the last GUI window is closed. If a message arrives while its function is still running due to a previous arrival of the same message, by default the function will not be called again; instead, the message will be treated as unmonitored. If this is undesirable, there are multiple ways it can be avoided: If the message is posted rather than sent and has a number greater than 0x0311, it can be buffered until its function completes by specifying Critical as the first line of the function. Alternatively, Thread Interrupt can achieve the same effect as long as it lasts long enough for the function to finish. Using Critical to increase the message check interval gives the function more time to complete before any messages are dispatched. An interval greater than 16 may be needed for reliability. Due to the granularity of the system timer (usually 15.6 milliseconds), the default interval for non-Critical threads (5 milliseconds) might appear to pass the instant after the function starts. Ensuring that the monitor function returns quickly reduces the risk that messages will be missed due to MaxThreads. One way to do this is to have it queue up a future thread by posting to its own script a monitored message number greater than 0x0311. That message's function should use Critical as its first line to ensure that its messages are buffered. Alternatively, a timer can be used to queue up a future thread. Specifying a higher value for the MaxThreads parameter allows the function to be interrupted to process the newly-received message. If a monitored message that is numerically greater than 0x0311 is posted while the script is uninterruptible, the message is buffered; that is, its function is not called until the script becomes interruptible. However, messages which are sent rather than posted cannot be buffered as they must provide a return value. Posted messages also might not be buffered when a modal message loop is running, such as for a system dialog, ListView drag-drop operation or menu. If a monitored message arrives and is not buffered, its function is called immediately even if the thread is uninterruptible when the message is received. The priority of OnMessage threads is always 0. Consequently, no messages are monitored or buffered when the current thread's priority is higher than 0. Caution should be used when monitoring system messages (those below 0x0400). For example, if a monitor function does not finish quickly, the response to the message might take longer than the system expects, which might cause side-effects. Unwanted behavior may also occur if a monitor function returns an integer to suppress further processing of a message, but the system expected different processing or a different response. When the script is displaying a system dialog such as MsgBox, any message posted to a control is not monitored. For example, if the script is displaying a message box and the user clicks a button in a GUI window, the WM_LBUTTONDOWN message is sent directly to the button without calling the monitor function. Although an external program may post messages directly to a script's thread via PostThreadMessage() or other API call, this is not recommended because the messages would be lost if the script is displaying a system window such as a message box. Instead, it is usually best to post or send the messages to the script's main window or one of its GUI windows. Related CallbackCreate, OnExit, OnClipboardChange, PostMessage, SendMessage, Functions, List of Windows Messages, Threads, Critical, DllCall Examples Monitors mouse clicks in a GUI window. Related topic: ContextMenu event MyGui := Gui(, "Example Window") MyGui.Add("Text",, "Click anywhere in this window.") MyGui.Add("Edit", "w200") MyGui.Show OnMessage 0x0201, WM_LBUTTONDOWN WM_LBUTTONDOWN(wParam, lParam, msg, hwnd) { X := lParam & 0xFFFF Y := lParam >> 16 Control := "" thisGui := GuiFromHwnd(hwnd) thisGuiControl := GuiCtrlFromHwnd(hwnd) if thisGuiControl { thisGui := thisGuiControl.Gui Control := "`n(in control " . thisGuiControl.ClassNN . ")" } ToolTip "You left-clicked in Gui window '" thisGui.Title "' at client coordinates " X "x" Y "." Control } Detects system shutdown/logoff and allows the user to abort it. On Windows Vista and later, the system displays a user interface showing which program is blocking shutdown/logoff and allowing the user to force shutdown/logoff. On older OSes, the script displays a confirmation prompt. Related topic: OnExit ; The following

DllCall is optional: it tells the OS to shut down this script first (prior to all other applications). DllCall("kernel32.dll\SetProcessShutdownParameters", "UInt", 0x4FF, "UInt", 0) OnMessage(0x0011, On_WM_QUERYENDSESSION) Persistent On_WM_QUERYENDSESSION(wParam, lParam, *) { ENDSESSION_LOGOFF := 0x80000000 if (lParam & ENDSESSION_LOGOFF) ; User is logging off. EventType := "Logoff" else ; System is either shutting down or restarting. EventType := "Shutdown" try { ; Set a prompt for the OS shutdown UI to display. We do not display ; our own confirmation prompt because we have only 5 seconds before ; the OS displays the shutdown UI anyway. Also, a program without ; a visible window cannot block shutdown without providing a reason. BlockShutdown("Example script attempting to prevent " EventType ".") return false } catch { ; ShutdownBlockReasonCreate is not available, so this is probably ; Windows XP, 2003 or 2000, where we can actually prevent shutdown. Result := MsgBox(EventType " in progress. Allow it?",, "YN") if (Result = "Yes") return true ; Tell the OS to allow the shutdown/logoff to continue. else return false ; Tell the OS to abort the shutdown/logoff. } } BlockShutdown(Reason) { ; If your script has a visible GUI, use it instead of A_ScriptHwnd. DllCall("ShutdownBlockReasonCreate", "ptr", A_ScriptHwnd, "wstr", Reason) OnExit StopBlockingShutdown } StopBlockingShutdown(*) { OnExit StopBlockingShutdown, 0 DllCall("ShutdownBlockReasonDestroy", "ptr", A_ScriptHwnd) } Receives a custom message and up to two numbers from some other script or program (to send strings rather than numbers, see the example after this one). OnMessage 0x5555, MsgMonitor Persistent MsgMonitor(wParam, lParam, msg, *) { ; Since returning quickly is often important, it is better to use ToolTip than ; something like MsgBox that would prevent the function from finishing: ToolTip "Message " msg " arrived:`nWPARAM: " wParam "`nLPARAM: " lParam } ; The following could be used inside some other script to run the function inside the above script: SetTitleMatchMode 2 DetectHiddenWindows True if WinExist ("Name of Receiving Script.ahk ahk_class AutoHotkey") PostMessage 0x5555, 11, 22 ; The message is sent to the "last found window" due to WinExist above. DetectHiddenWindows False ; Must not be turned off until after PostMessage. Sends a string of any length from one script to another. To use this, save and run both of the following scripts then press Win+Space to show an input box that will prompt you to type in a string. Both scripts must use the same native encoding. Save the following script as Receiver.ahk then launch it. #SingleInstance OnMessage 0x004A, Receive_WM_COPYDATA ; 0x004A is WM_COPYDATA Persistent Receive_WM_COPYDATA(wParam, lParam, msg, hwnd) { StringAddress := NumGet(lParam, 2*A_PtrSize, "Ptr") ; Retrieves the CopyDataStruct's lpData member. CopyOfData := StrGet(StringAddress) ; Copy the string out of the structure. ; Show it with ToolTip vs. MsgBox so we can return in a timely fashion: ToolTip A_ScriptName "`nReceived the following string:`n" CopyOfData return true ; Returning 1 (true) is the traditional way to acknowledge this message. } Save the following script as Sender.ahk then launch it. After that, press the Win+Space hotkey. TargetScriptTitle := "Receiver.ahk ahk_class AutoHotkey" #space:: ; Win+Space hotkey. Press it to show an input box for entry of a message string. { ib := InputBox("Enter some text to Send:", "Send text via WM_COPYDATA") if ib.Result = "Cancel" ; User pressed the Cancel button. return result := Send_WM_COPYDATA(ib.Value, TargetScriptTitle) if result = "" MsgBox "SendMessage failed or timed out. Does the following WinTitle exist?:`n" TargetScriptTitle else if (result = 0) MsgBox "Message sent but the target window responded with 0, which may mean it ignored it." } Send_WM_COPYDATA(StringToSend, TargetScriptTitle) ; This function sends the specified string to the specified window and returns the reply. ; The reply is 1 if the target window processed the message, or 0 if it ignored it. { CopyDataStruct := Buffer(3*A_PtrSize) ; Set up the structure's memory area. ; First set the structure's cbData member to the size of the string, including its zero terminator: SizeInBytes := (StrLen(StringToSend) + 1) * 2 NumPut("Ptr", SizeInBytes ; OS requires that this be done. , "Ptr", StrPtr(StringToSend) ; Set lpData to point to the string itself. , CopyDataStruct, A_PtrSize) Prev_DetectHiddenWindows := A_DetectHiddenWindows Prev_TitleMatchMode := A_TitleMatchMode DetectHiddenWindows True SetTitleMatchMode 2 TimeOutTime := 4000 ; Optional. Milliseconds to wait for response from receiver.ahk. Default is 5000 ; Must use SendMessage not PostMessage. RetValue := SendMessage(0x004A, 0, CopyDataStruct,, TargetScriptTitle,,,, TimeOutTime) ; 0x004A is WM_COPYDATA. DetectHiddenWindows Prev_DetectHiddenWindows ; Restore original setting for the caller. SetTitleMatchMode Prev_TitleMatchMode ; Same. return RetValue ; Return SendMessage's reply back to our caller. } See the WinLIRC client script for a demonstration of how to use OnMessage to receive a notification when data has arrived on a network connection. Ord - Syntax & Usage | AutoHotkey v2 Ord Returns the ordinal value (numeric character code) of the first character in the specified string. Number := Ord(String) Parameters String Type: String The string whose ordinal value is retrieved. Return Value Type: Integer This function returns the ordinal value (numeric character code) of the first character in the specified string. If the string is empty, zero is returned. If the string begins with a Unicode supplementary character, this function returns the corresponding Unicode character code (a number between 0x10000 and 0x10FFFF). Otherwise it returns a value in the range 0 to 0xFF (for ANSI) or 0 to 0xFFFF (for Unicode). See Unicode vs ANSI for details. Related Chr Examples Both message boxes below show 116, because only the first character is considered. MsgBox Ord("t") MsgBox Ord("test") OutputDebug - Syntax & Usage | AutoHotkey v2 OutputDebug Sends a string to the debugger (if any) for display. OutputDebug Text Parameters Text Type: String The text to send to the debugger for display. This text may include linefeed characters (`n) to start new lines. In addition, a single long line can be broken up into several shorter ones by means of a continuation section. Remarks If the script's process has no debugger, the system debugger displays the string. If the system debugger is not active, this function has no effect. One example of a debugger is DebugView, which is free and available at microsoft.com. See also: other debugging methods Related FileAppend, continuation sections Examples Sends a string to the debugger (if any) for display. OutputDebug A_Now ': Because the window "' TargetWindowTitle '" did not exist, the process was aborted.' Pause - Syntax & Usage | AutoHotkey v2 Pause Pauses the script's current thread. Pause NewState Parameters NewState Type: Integer or String If omitted, the current thread is paused. Otherwise, specify one of the following values: 1 or True: Marks the thread beneath the current thread as paused so that when it resumes, it will finish the function it was running (if any) and then enter a paused state. If there is no thread beneath the current thread, the script itself is paused, which prevents timers from running (this effect is the same as having used the menu item "Pause Script" while the script has no threads). 0 or False: Unpauses the underlying thread. -1: Toggles the pause state of the underlying thread. Remarks A_IsPaused contains the pause state of the underlying thread. By default, the script can also be paused via its tray icon or main window. Unlike Suspend -- which disables hotkeys and hotstrings -- turning on pause will freeze the thread (the current thread if NewState was omitted, otherwise the underlying thread). As a side-effect, any interrupted threads beneath it will lie dormant. Whenever any thread is paused, timers will not run. By contrast, explicitly launched threads such as hotkeys and menu items can still be launched; but when their threads finish, the underlying thread will still be paused. In other words, each thread can be paused independently of the others. With the default icons, the color of the tray icon changes from green to red whenever the script's current thread is in a paused state. This color change can be avoided by freezing the icon, which is achieved by specifying 1 for the last parameter of TraySetIcon. For example: TraySetIcon ,, 1 To disable timers without pausing the script, use Thread "NoTimers". A script is always halted (though not officially paused) while it is displaying any kind of menu (tray menu, menu bar, GUI context menu, etc.) Related Suspend, Menu object, ExitApp, Threads, SetTimer Examples Use Pause to halt the script, such as to inspect variables. ListVars Pause ExitApp ; This line will not execute until the user unpauses the script. Press a hotkey once to pause the script. Press it again to unpause. Pause::Pause -1 ; The Pause/Break key. #p::Pause -1 ; Win+P Sends a Pause command to another script. DetectHiddenWindows True WM_COMMAND := 0x0111 ID_FILE_PAUSE := 65403 PostMessage WM_COMMAND, ID_FILE_PAUSE,,, "C:\YourScript.ahk ahk_class AutoHotkey" Persistent - Syntax & Usage | AutoHotkey v2 Persistent Prevents the script from exiting automatically when its last thread completes, allowing it to stay running in an idle state. Persistent Persist Parameters Persist Type: Boolean If true or omitted, the script will be kept running after all threads have exited, even if none of the other conditions for keeping the script running are met. If false, the default behaviour is restored. Return Value Type: Integer (boolean) This function returns the previous value of the global setting; either 0 (false, the default value) or 1 (true). Remarks If the script is persistent, it will stay running after startup completes and all other threads have exited. It is usually unnecessary to call this function because the script is automatically persistent in most of the common cases where the user would want it to keep running, such as to respond to hotkeys, execute timers or display a GUI. Some cases where this function might be needed (if it is intended to stay running when there are no running threads or hotkeys, timers, etc.) include: Scripts which use OnMessage or CallbackCreate and DllCall to respond to events, since those functions don't make the script persistent. Scripts that are executed by selecting a custom tray menu item. Scripts which create or retrieve COM objects and use ComObjConnect to respond to the object's events. If this function is added to an existing script, some or all occurrences of Exit might need to be changed to ExitApp. This is because Exit will not terminate a persistent script; it terminates only the current thread. Related Exit, ExitApp Examples Prevent the script from exiting automatically. ; This script will not exit automatically, even though it has nothing to do. ; However, you can use its tray icon to open the script in an editor, or to ; launch Window Spy or the Help file. Persistent PixelGetColor - Syntax & Usage | AutoHotkey v2 PixelGetColor Retrieves the color of the pixel at the specified x,y coordinates. Color := PixelGetColor(X, Y , Mode) Parameters X, Y Type: Integer The X and Y coordinates of the pixel. Coordinates are relative to the active window's client area unless CoordMode was used to change that. Mode Type: String This parameter may contain zero or more of the following words. If more than one word is present, separate each from the next with a space (e.g. "Alt Slow"). Alt: Uses an alternate method to retrieve the color, which should be used when the normal method produces invalid or inaccurate colors for a particular type of window. This method is about 10% slower than the normal method. Slow: Uses a more elaborate method to retrieve the color, which may work in certain full-screen applications when the other methods fail. This method is about three times slower than the normal method. Note: Slow takes precedence over Alt, so there is no need to specify Alt in this case. Return Value Type: String This function returns a hexadecimal numeric string representing the RGB (red-green-blue) color of the pixel. For example, the color purple is defined 0x800080 because it has an intensity of 0x80 (128) for its blue and red components but an intensity of 0x00 (0) for its green component. Error Handling An OSError is thrown on failure. Remarks The pixel must be visible; in other words, it is not possible to retrieve the pixel color of a window hidden behind another window. By contrast, pixels beneath the mouse cursor can usually be detected. The exception to this is game cursors, which in most cases will hide any pixels beneath them. Use Window Spy (available in tray icon menu) or the example at the bottom of this page to determine the colors currently on the screen. Known limitations: A window that is partially transparent or that has one of its colors marked invisible (WinSetTransColor) typically yields colors for the window behind itself rather than its own. PixelGetColor might not produce accurate results for certain applications. If this occurs, try

specifying the word Alt or Slow in the last parameter. Related PixelSearch, ImageSearch, CoordMode, MouseGetPos Examples Press a hotkey to show the color of the pixel located at the current position of the mouse cursor. ^!z:: ; Control+Alt+Z hotkey. { MouseGetPos &MouseX, &MouseY MsgBox "The color at the current cursor position is " PixelGetColor(MouseX, MouseY) } PixelSearch - Syntax & Usage | AutoHotkey v2 PixelSearch Searches a region of the screen for a pixel of the specified color. PixelSearch &OutputVarX, &OutputVarY, X1, Y1, X2, Y2, ColorID , Variation Parameters &OutputVarX, &OutputVarY Type: VarRef References to the output variables in which to store the X and Y coordinates of the first pixel that matches ColorID (if no match is found, the variables are made blank). Coordinates are relative to the active window's client area unless CoordMode was used to change that. X1, Y1 Type: Integer The X and Y coordinates of the starting corner of the rectangle to search. Coordinates are relative to the active window's client area unless CoordMode was used to change that. X2, Y2 Type: Integer The X and Y coordinates of the ending corner of the rectangle to search. Coordinates are relative to the active window's client area unless CoordMode was used to change that. ColorID Type: Integer The color ID to search for. This is typically expressed as a hexadecimal number in red-green-blue (RGB) format. For example: 0x9d6346. Color IDs can be determined using Window Spy (accessible from the tray menu) or via PixelGetColor. Variation Type: Integer A number between 0 and 255 (inclusive) to indicate the allowed number of shades of variation in either direction for the intensity of the red, green, and blue components of the color. For example, if 2 is specified and ColorID is 0x444444, any color from 0x424242 to 0x464646 will be considered a match. This parameter is helpful if the color sought is not always exactly the same shade. If you specify 255 shades of variation, all colors will match. The default is 0 shades. Return Value Type: Integer (boolean) This function returns 1 (true) if the color was found in the specified region, or 0 (false) if it was not found. Error Handling An OSError is thrown if there was a problem that prevented the function from conducting the search. Remarks The region to be searched must be visible; in other words, it is not possible to search a region of a window hidden behind another window. By contrast, pixels beneath the mouse cursor can usually be detected. The exception to this is cursors in games, which in most cases will hide any pixels beneath them. Although color depths as low as 8-bit (256-color) are supported, the fast mode performs much better in 24-bit or 32-bit color. The search starts at the coordinates specified by X1 and Y1 and checks all pixels in the row from X1 to X2 for a match. If no match is found there, the search continues toward Y2, row by row, until it finds a matching pixel. The search order depends on the order of the parameters. In other words, if X1 is greater than X2, the search will be conducted from right to left, starting at column X1. Similarly, if Y1 is greater than Y2, the search will be conducted from bottom to top. If the region to be searched is large and the search is repeated with high frequency, it may consume a lot of CPU time. To alleviate this, keep the size of the area to a minimum. Related PixelGetColor, ImageSearch, CoordMode, MouseGetPos Examples Searches a region of the active window for a pixel and stores in Px and Py the X and Y coordinates of the first pixel that matches the specified color with 3 shades of variation. if PixelSearch(&Px, &Py, 200, 200, 300, 300, 0x9d6346, 3) MsgBox "A color within 3 shades of variation was found at X" Px " Y" Py else MsgBox "That color was not found in the specified region." PostMessage - Syntax & Usage | AutoHotkey v2 PostMessage Places a message in the message queue of a window or control. PostMessage Msg , wParam, lParam, Control, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Msg Type: Integer The message number to send. See the message list to determine the number. wParam, lParam Type: Integer The message parameters. If omitted, each parameter defaults to 0. Each parameter must be an integer. If AutoHotkey or the target window is 32-bit, only the parameter's low 32 bits are used; that is, values are truncated if outside the range -2147483648 to 2147483647 (-0x80000000 to 0x7FFFFFFF) for signed values, or 0 to 4294967295 (0xFFFFFFFF) for unsigned values. If AutoHotkey and the target window are both 64-bit, any integer value supported by AutoHotkey can be used. Control Type: String, Integer or Object If this parameter is omitted, the message will be posted directly to the target window rather than one of its controls. Otherwise, this parameter can be the control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. If this parameter specifies a HWND (as an integer or object), it is not required to be the HWND of a control (child window). That is, it can also be the HWND of a top-level window. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window or control could not be found. An OSError is thrown if the message could not be posted. For example, if the target window is running at a higher integrity level than the script (i.e. it is running as admin while the script is not), messages may be blocked. Remarks This function should be used with caution because sending a message to the wrong window (or sending an invalid message) might cause unexpected behavior or even crash the target application. This is because most applications are not designed to expect certain types of messages from external sources. PostMessage places the message in the message queue associated with the target window. It does not wait for acknowledgement or reply. By contrast, SendMessage waits for the target window to process the message, up until the timeout period expires. Unlike SendMessage, PostMessage usually only sends basic numeric values, not pointers to structures or strings. To send a message to all windows in the system, including those that are hidden or disabled, specify 0xFFFF for WinTitle (0xFFFF is HWND_BROADCAST). This technique should be used only for messages intended to be broadcast. To have a script receive a message, use OnMessage. See the Message Tutorial for an introduction to using this function. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SendMessage, Message List, Message Tutorial, OnMessage, Automating Winamp, DllCall, ControlSend, MenuSelect Examples Switches the active window's keyboard layout/language to English (US). PostMessage 0x0050, 0, 0x4090409,, "A" ; 0x0050 is WM_INPUTLANGCHANGEREQUEST. List of Process Functions | AutoHotkey v2 Process Functions Functions for performing the following operations on a process: check if it exists; change its priority; close it; wait for it to exist; wait for it to close. Click on a function name for details. Function Description ProcessClose Forces the first matching process to close. ProcessExist Checks if the specified process exists. ProcessGetName Returns the name of the specified process. ProcessGetParent Returns the process ID (PID) of the process which created the specified process. ProcessGetPath Returns the path of the specified process. ProcessSetPriority Changes the priority level of the first matching process. ProcessWait Waits for the specified process to exist. ProcessWaitClose Waits for all matching processes to close. Remarks Process list: Although there is no ProcessList function, example #1 and example #2 demonstrate how to retrieve a list of processes via DllCall or COM. Related Run, WinClose, WinKill, WinWait, WinWaitClose, WinExist, Win functions Examples Retrieves a list of running processes via DllCall then shows them in a message box. d := " | " ; string separator s := 4096 ; size of buffers and arrays (4 KB) ScriptPID := ProcessExist() ; The PID of this running script. ; Get the handle of this script with PROCESS_QUERY_INFORMATION (0x0400): h := DllCall("OpenProcess", "UInt", 0x0400, "Int", false, "UInt", ScriptPID, "Ptr") ; Open an adjustable access token with this process (TOKEN_ADJUST_PRIVILEGES = 32): DllCall ("Advapi32.dll\OpenProcessToken", "Ptr", h, "UInt", 32, "PtrP", &t := 0) ; Retrieves the locally unique identifier of the debug privilege: DllCall ("Advapi32.dll\LookupPrivilegeValue", "Ptr", 0, "Str", "SeDebugPrivilege", "Int64P", &luid := 0) ti := Buffer(16, 0) ; structure of privileges NumPut("UInt", 1 ; one entry in the privileges array... , "Int64", luid , "UInt", 2 ; Enable this privilege: SE_PRIVILEGE_ENABLED = 2 , ti) ; Update the privileges of this process with the new access token: r := DllCall("Advapi32.dll\AdjustTokenPrivileges", "Ptr", t, "Int", false, "Ptr", ti, "UInt", 0, "Ptr", 0, "Ptr", 0) DllCall ("CloseHandle", "Ptr", t) ; Close this access token handle to save memory. DllCall("CloseHandle", "Ptr", h) ; Close this process handle to save memory. hModule := DllCall("LoadLibrary", "Str", "Psapi.dll") ; Increase performance by preloading the library. a := Buffer(s) ; An array that receives the list of process identifiers: c := 0 ; counter for process idendifiers l := "" DllCall("Psapi.dll\EnumProcesses", "Ptr", a, "UInt", s, "UIntP", &r) Loop r // 4 ; Parse array for identifiers as DWORDs (32 bits): { id := NumGet(a, A_Index * 4, "UInt") ; Open process with: PROCESS_VM_READ (0x0010) | PROCESS_QUERY_INFORMATION (0x0400) h := DllCall("OpenProcess", "UInt", 0x0010 | 0x0400, "Int", false, "UInt", id, "Ptr") if !h continue n := Buffer(s, 0) ; a buffer that receives the base name of the module: e := DllCall("Psapi.dll\GetModuleBaseName", "Ptr", h, "Ptr", 0, "Ptr", n, "UInt", s//2) if !e ; fall-back method for 64-bit processes when in 32-bit mode: e := DllCall("Psapi.dll\GetProcessImageFileName", "Ptr", h, "Ptr", n, "UInt", s//2) SplitPath StrGet(n), &n DllCall("CloseHandle", "Ptr", h) ; close process handle to save memory if (n && e) ; if image is not null add to list: l .= n "`n", c++ } DllCall("FreeLibrary", "Ptr", hModule) ; Unload the library to free memory. ;l := Sort(l) ; Uncomment this line to sort the list alphabetically. MsgBox StrReplace(l, "`n", d), c " Processes", 0 Retrieves a list of running processes via COM. For Win32_Process, see Microsoft Docs. MyGui := Gui(, "Process List") LV := MyGui.Add("ListView", "x2 y0 w400 h500", ["Process Name","Command Line"]) for process in ComObjGet("winmgmts:").ExecQuery("Select * from Win32_Process") LV.Add("", process.Name, process.CommandLine) MyGui.Show ProcessClose - Syntax & Usage | AutoHotkey v2 ProcessClose Forces the first matching process to close. ProcessClose PIDOrName Parameters PIDOrName Type: Integer or String Specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. ProcessExist can also be used to discover a PID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Return Value Type: Integer This function returns the Process ID (PID) of the specified process. If a matching process is not found or cannot be manipulated, zero is returned. Remarks Since the process will be abruptly terminated -- possibly interrupting its work at a critical point or resulting in the loss of unsaved data in its windows (if it has any) -- this function should be used only if a process cannot be closed by using WinClose on one of its windows. Related Run, WinClose, WinKill, Process functions, Win functions Examples Forces Notepad to close (be warned that any unsaved data will be lost). ProcessClose "notepad.exe" ProcessExist - Syntax & Usage | AutoHotkey v2 ProcessExist Checks if the specified process exists. PID := ProcessExist(PIDOrName) Parameters PIDOrName Type: Integer or String If blank or omitted, the script's own process is used. Otherwise, specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple

running processes, only the first process will be operated upon. The name is not case sensitive. Return Value Type: Integer This function returns the Process ID (PID) of the specified process. If there is no matching process, zero is returned. Related Run, WinExist, Process functions, Win functions Examples Checks if a process of Notepad exists. if (PID := ProcessExist("notepad.exe")) MsgBox "Notepad exists and has the Process ID " PID "." else MsgBox "Notepad does not exist." ProcessGetName / ProcessGetPath - Syntax & Usage | AutoHotkey v2 ProcessGetName / ProcessGetPath Returns the name or path of the specified process. Name := ProcessGetName(PIDOrName) Path := ProcessGetPath(PIDOrName) Parameters PIDOrName Type: Integer or String If blank or omitted, the script's own process is used. Otherwise, specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Return Value Type: String ProcessGetName returns the name of the specified process. For example: notepad.exe. ProcessGetPath returns the path of the specified process. For example: C:\Windows\notepad.exe. Error Handling A TargetError is thrown if the process could not be found. An OSError is thrown if the name/path could not be retrieved. Related Process functions, Run, WinGetProcessName, WinGetProcessPath Examples Get the name and path of a process used to open a document. Run "license.rtf",,, &pid ; This is likely to exist in C:\Windows\System32. try { name := ProcessGetName(pid) path := ProcessGetPath(pid) } MsgBox "Name: " (name ?? "could not be retrieved") "`n" . "Path: " (path ?? "could not be retrieved") ProcessGetParent - Syntax & Usage | AutoHotkey v2 ProcessGetParent Returns the process ID (PID) of the process which created the specified process. PID := ProcessGetParent(PIDOrName) Parameters PIDOrName Type: Integer or String If blank or omitted, the script's own process is used. Otherwise, specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Return Value Type: Integer This function returns the process ID (PID) of the process which created the specified process. Error Handling A TargetError is thrown if the specified process could not be found. Remarks If the parent process is no longer running, there is some risk that the returned PID has been reused by the system, and now identifies a different process. Related Process functions Examples Display the name of the process which launched the script. try MsgBox ProcessGetName(ProcessGetParent()) catch MsgBox "Unable to retrieve parent process name; the process has likely exited." ProcessSetPriority - Syntax & Usage | AutoHotkey v2 ProcessSetPriority Changes the priority level of the first matching process. ProcessSetPriority Level , PIDOrName Parameters Level Type: String Specify one of the following words or letters: Low (or L) BelowNormal (or B) Normal (or N) AboveNormal (or A) High (or H) Realtime (or R) Note that any process not designed to run at Realtime priority might reduce system stability if set to that level. PIDOrName Type: Integer or String If blank or omitted, the script's own process is used. Otherwise, specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. ProcessExist can also be used to discover a PID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Return Value Type: Integer This function returns the Process ID (PID) of the specified process. If a matching process is not found or cannot be manipulated, zero is returned. Remarks The current priority level of a process can be seen in the Windows Task Manager. Related Run, Process functions, Win functions Examples Launches Notepad, sets its priority to "High" and shows its current PID. Run "notepad.exe", , , &NewPID ProcessSetPriority "High", NewPID MsgBox "The newly launched Notepad's PID is " NewPID Press a hotkey to change the priority of the active window's process. #z:: ; Win+Z hotkey { active_pid := WinGetPID("A") active_title := WinGetTitle("A") MyGui := Gui(, "Set Priority") MyGui.Add ("Text",, " (Press ESCAPE to cancel, or double-click a new priority level for the following window:)") MyGui.Add("Text", "wp", active_title) LB := MyGui.Add ("ListBox", "r5 Choose1", ["Normal","High","Low","BelowNormal","AboveNormal"]) LB.OnEvent("DoubleClick", SetPriority) MyGui.Add("Button", "default", "OK").OnEvent("Click", SetPriority) MyGui.OnEvent("Escape", (*) => MyGui.Destroy()) MyGui.OnEvent("Close", (*) => MyGui.Destroy()) MyGui.Show() SetPriority(*) { MyGui.Destroy() if ProcessSetPriority(LB.Text, active_pid) MsgBox "Success: Its priority was changed to " LB.Text else MsgBox "Error: Its priority could not be changed to " LB.Text } } ProcessWait - Syntax & Usage | AutoHotkey v2 ProcessWait Waits for the specified process to exist. PID := ProcessWait(PIDOrName , Timeout) Parameters PIDOrName Type: Integer or String Specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. ProcessExist can also be used to discover a PID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Timeout Type: Integer or Float If omitted, this function will wait indefinitely. Otherwise, specify the number of seconds (can contain a decimal point) to wait before timing out. Return Value Type: Integer If the specified process is discovered, this function returns the Process ID (PID) of the process. If the function times out, zero is returned. Remarks Processes are checked every 100 milliseconds; the moment the condition is satisfied, the function stops waiting. In other words, rather than waiting for the timeout to expire, it immediately returns and continues execution of the script. Also, while the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. Related ProcessWaitClose, Run, WinWait, Process functions, Win functions Examples Waits up to 5.5 seconds for Notepad to appear. If Notepad appears within this number of seconds, its priority is set to "Low" and the script's own priority is set to "High". After that, Notepad will be closed and a message box will be shown if it did not close within 5 seconds. NewPID := ProcessWait("notepad.exe", 5.5) if not NewPID { MsgBox "The specified process did not appear within 5.5 seconds." return } ; Otherwise: MsgBox "A matching process has appeared (Process ID is " NewPID ")." ProcessSetPriority "Low", NewPID ProcessSetPriority "High" ; Have the script set itself to high priority. WinClose "Untitled - Notepad" WaitPID := ProcessWaitClose(NewPID, 5) if WaitPID ; The PID still exists. MsgBox "The process did not close within 5 seconds." ProcessWaitClose - Syntax & Usage | AutoHotkey v2 ProcessWaitClose Waits for all matching processes to close. PID := ProcessWaitClose(PIDOrName , Timeout) Parameters PIDOrName Type: Integer or String Specify either a number (the PID) or a process name: PID: The Process ID, which is a number that uniquely identifies one specific process (this number is valid only during the lifetime of that process). The PID of a newly launched process can be determined via the Run function. Similarly, the PID of a window can be determined with WinGetPID. ProcessExist can also be used to discover a PID. Name: The name of a process is usually the same as its executable (without path), e.g. notepad.exe or winword.exe. Since a name might match multiple running processes, only the first process will be operated upon. The name is not case sensitive. Timeout Type: Integer or Float If omitted, this function will wait indefinitely. Otherwise, specify the number of seconds (can contain a decimal point) to wait before timing out. Return Value Type: Integer If all matching processes are closed, zero is returned. If this function times out, it returns the Process ID (PID) of the first matching process that still exists. Remarks Processes are checked every 100 milliseconds; the moment the condition is satisfied, the function stops waiting. In other words, rather than waiting for the timeout to expire, it immediately returns and continues execution of the script. Also, while the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. Related ProcessWait, Run, WinWaitClose, Process functions, Win functions Examples See example #1 on the ProcessWait page. Random - Syntax & Usage | AutoHotkey v2 Random Generates a pseudo-random number. N := Random(A, B) Parameters A, B Type: Integer or Float The minimum and/or maximum number to be generated, specified in either order. If only one parameter is specified, the other parameter defaults to 0. If both are omitted, the default is 0.0 to 1.0. For integers, the minimum value and maximum value are both included in the set of possible numbers that may be returned. The full range of 64-bit integers is supported. For floating point numbers, the maximum value is generally excluded. Return Value Type: Integer or Float This function returns a pseudo-randomly generated number, which is a number that simulates a true random number but is really a number based on a complicated formula to make determination/guessing of the next number extremely difficult. If either A or B is a floating point number or both are omitted, the result will be a floating point number. Otherwise, the result will be an integer. Remarks All numbers within the specified range have approximately the same probability of being generated. Although the specified maximum value is excluded by design, it may in theory be returned due to floating point rounding errors. This has not been confirmed, and might only be possible if the chosen bounds are larger than 2**53. Also note that since there may be up to 2**53 possible values (such as in the range 0.0 to 1.0), the probability of generating exactly the lower bound is generally very low. Examples Generates a random integer in the range 1 to 10 and stores it in N. N := Random(1, 10) Generates a random integer in the range 0 to 9 and stores it in N. N := Random(9) Generates a random floating point number in the range 0.0 to 1.0 and stores it in fraction. fraction := Random (0.0, 1.0) fraction := Random() ; Equivalent to the line above. RegCreateKey - Syntax & Usage | AutoHotkey v2 RegCreateKey Creates a registry key without writing a value. RegCreateKey KeyName Parameters KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a slash, as in this example: \\workstation01\HKEY_LOCAL_MACHINE KeyName can be omitted only if a registry loop is running, in which case it defaults to the key of the current loop item (even if the key has been deleted during the loop). If the item is a subkey, the full name of that subkey is used by default. Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks If KeyName specifies an existing registry key, RegCreateKey verifies that the script has write access to the key, but makes no changes. Otherwise, RegCreateKey attempts to create the key (along with its ancestors, if necessary). For details about how to access the registry of a remote computer, see the remarks in registry loop. To create subkeys in the 64-bit sections of the registry in a 32-bit script or vice versa, use SetRegView. Related

RegDelete, RegDeleteKey, RegRead, RegWrite, Registry-loop, SetRegView Examples Creates an empty registry key. If Notepad++ is installed, this has the effect of adding it to the "open with" menu for .ahk files. RegCreateKey "HKCU\Software\Classes\.ahk\OpenWithList\notepad++.exe" RegDelete - Syntax & Usage | AutoHotkey v2 RegDelete Deletes a value from the registry. RegDelete KeyName, ValueName Parameters KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a slash, as in this example: \\workstation01\HKEY_LOCAL_MACHINE KeyName can be omitted only if a registry loop is running, in which case it defaults to the key of the current loop item. If the item is a subkey, the full name of that subkey is used by default. If the item is a value, ValueName defaults to the name of that value, but can be overridden. ValueName Type: String The name of the value to delete. If blank or omitted, the key's default value will be deleted (except as noted above). The default value is displayed as "(Default)" by RegEdit. Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks Warning: Deleting from the registry is potentially dangerous - please exercise caution! To retrieve and operate upon multiple registry keys or values, consider using a registry loop. Within a registry loop, RegDelete does not necessarily delete the current loop item. If the item is a subkey, RegDelete() only deletes its default value. For details about how to access the registry of a remote computer, see the remarks in registry loop. To delete entries from the 64-bit sections of the registry in a 32-bit script or vice versa, use SetRegView. Related RegCreateKey, RegDeleteKey, RegRead, RegWrite, Registry-loop, SetRegView, IniDelete Examples Deletes a value from the registry. RegDelete "HKEY_LOCAL_MACHINE\Software\SomeApplication", "TestValue" RegDeleteKey - Syntax & Usage | AutoHotkey v2 RegDeleteKey Deletes a subkey from the registry. RegDeleteKey KeyName Parameters KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a slash, as in this example: \\workstation01\HKEY_LOCAL_MACHINE KeyName can be omitted only if a registry loop is running, in which case it defaults to the key of the current loop item. If the item is a subkey, the full name of that subkey is used by default. Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks Warning: Deleting from the registry is potentially dangerous - please exercise caution! To retrieve and operate upon multiple registry keys or values, consider using a registry loop. Within a registry loop, RegDeleteKey does not necessarily delete the current loop item. If the item is a subkey, RegDeleteKey() deletes the key itself. If the item is a value, RegDeleteKey() deletes the key which contains that value, including all subkeys and values. For details about how to access the registry of a remote computer, see the remarks in registry loop. To delete entries from the 64-bit sections of the registry in a 32-bit script or vice versa, use SetRegView. Related RegCreateKey, RegDelete, RegRead, RegWrite, Registry-loop, SetRegView, IniDelete Examples Deletes a subkey from the registry. RegDeleteKey "HKEY_LOCAL_MACHINE\Software\SomeApplication" RegExMatch - Syntax & Usage | AutoHotkey v2 RegExMatch Determines whether a string contains a pattern (regular expression). FoundPos := RegExMatch(Haystack, NeedleRegEx , &OutputVar, StartingPos) Parameters Haystack Type: String The string whose content is searched. This may contain binary zero. NeedleRegEx Type: String The pattern to search for, which is a Perl-compatible regular expression (PCRE). The pattern's options (if any) must be included at the beginning of the string followed by a close-parenthesis. For example, the pattern i)abc.*123 would turn on the case-insensitive option and search for "abc", followed by zero or more occurrences of any character, followed by "123". If there are no options, the ")" is optional; for example,)abc is equivalent to abc. Although NeedleRegEx cannot contain binary zero, the pattern \x00 can be used to match a binary zero within Haystack. &OutputVar Type: VarRef Specify a reference to a output variable in which to store a match object, which can be used to retrieve the position, length and value of the overall match and of each captured subpattern, if any are present. If the pattern is not found (that is, if the function returns 0), this variable is made blank. StartingPos Type: Integer If StartingPos is omitted, it defaults to 1 (the beginning of Haystack). Otherwise, specify 2 to start at the second character, 3 to start at the third, and so on. If StartingPos is beyond the length of Haystack, the search starts at the empty string that lies at the end of Haystack (which typically results in no match). Specify a negative StartingPos to start at that position from the right. For example, -1 starts at the last character and -2 starts at the next-to-last character. If StartingPos tries to go beyond the left end of Haystack, all of Haystack is searched. Specify 0 to start at the end of Haystack; i.e. the position to the right of the last character. This can be used with zero-width assertions such as (?<=a). Regardless of the value of StartingPos, the return value is always relative to the first character of Haystack. For example, the position of "abc" in "123abc789" is always 4. Return Value Type: Integer This function returns the position of the leftmost occurrence of NeedleRegEx in the string Haystack. Position 1 is the first character. Zero is returned if the pattern is not found. Errors Syntax errors: If the pattern contains a syntax error, an Error is thrown with a message in the following form: Compile error N at offset M: description. In that string, N is the PCRE error number, M is the position of the offending character inside the regular expression, and description is the text describing the error. Execution errors: If an error occurs during the execution of the regular expression, an Error is thrown. The Extra property of the error object contains the PCRE error number. Although such errors are rare, the ones most likely to occur are "too many possible empty-string matches" (-22), "recursion too deep" (-21), and "reached match limit" (-8). If these happen, try to redesign the pattern to be more restrictive, such as replacing each * with a ?, +, or a limit like {0,3} wherever feasible. Options See Options for modifiers such as i)abc, which turns off case-sensitivity in the pattern "abc". Match Object (RegExMatchInfo) If a match is found, an object containing information about the match is stored in OutputVar. This object has the following methods and properties: Match.Pos, Match.Pos[N] or Match.Pos(N): Returns the position of the overall match or a captured subpattern. Match.Len, Match.Len[N] or Match.Len(N): Returns the length of the overall match or a captured subpattern. Match.Name[N] or Match.Name(N): Returns the name of the given subpattern, if it has one. Match.Count: Returns the overall number of subpatterns (capturing groups), which is also the maximum value for N. Match.Mark: Returns the NAME of the last encountered (*MARK:NAME), when applicable. Match[] or Match[N]: Returns the overall match or a captured subpattern. All of the above allow N to be any of the following: 0 for the overall match. The number of a subpattern, even one that also has a name. The name of a subpattern. Match.N: Shorthand for Match["N"], where N is any unquoted name or number which does not conflict with a defined property (listed above). For example, match.1 or match.Year. The object also supports enumeration; that is, the for-loop is supported. Alternatively, use Loop Match.Count. Performance To search for a simple substring inside a larger string, use InStr because it is faster than RegExMatch. To improve performance, the 100 most recently used regular expressions are kept cached in memory (in compiled form). The study option (S) can sometimes improve the performance of a regular expression that is used many times (such as in a loop). Remarks A subpattern may be given a name such as the word Year in the pattern (?P\d{4}). Such names may consist of up to 32 alphanumeric characters and underscores. Note that named subpatterns are also numbered, so if an unnamed subpattern occurs after "Year", it would be stored in OutputVar[2], not OutputVar[1]. Most characters like abc123 can be used literally inside a regular expression. However, the characters \.*?+[{|()^$ must be preceded by a backslash to be seen as literal. For example, \. is a literal period and \\ is a literal backslash. Escaping can be avoided by using \Q...\E. For example: \QLiteral Text\E. Within a regular expression, special characters such as tab and newline can be escaped with either an accent (`) or a backslash (\). For example, `t is the same as \t except when the x option is used. To learn the basics of regular expressions (or refresh your memory of pattern syntax), see the RegEx Quick Reference. AutoHotkey's regular expressions are implemented using Perl-compatible Regular Expressions (PCRE) from www.pcre.org. Within an expression, the a ~= b can be used as shorthand for RegExMatch(a, b). Related RegExReplace, RegEx Quick Reference, Regular Expression Callouts, InStr, SubStr, SetTitleMatchMode RegEx, Global matching and Grep (forum link) Common sources of text data: FileRead, Download, A_Clipboard, GUI Edit controls Examples For general RegEx examples, see the RegEx Quick Reference. Reports 4, which is the position where the match was found. MsgBox RegExMatch("xxxabc123xyz", "abc.*xyz") Reports 7 because the $ requires the match to be at the end. MsgBox RegExMatch("abc123123", "123$") Reports 1 because a match was achieved via the case-insensitive option. MsgBox RegExMatch("abc123", "i)^ABC") Reports 1 and stores "XYZ" in SubPat[1]. MsgBox RegExMatch("abcXYZ123", "abc(.*)123", &SubPat) Reports 7 instead of 1 due to the starting position 2 instead of 1. MsgBox RegExMatch("abc123abc456", "abc\d+",, 2) Demonstrates the usage of the Match object. FoundPos := RegExMatch("Michiganroad 72", "(.*) (?\d+)", &SubPat) MsgBox SubPat.Count ": " SubPat[1] " " SubPat.Name[2] "=" SubPat.nr ; Displays "2: Michiganroad nr=72" Retrieves the extension of a file. Note that SplitPath can also be used for this, which is more reliable. Path := "C:\Foo\Bar\Baz.txt" RegExMatch(Path, "\w+$", &Extension) MsgBox Extension[] ; Reports "txt". Similar to AutoHotkey v1's Transform Deref, the following function expands variable references and escape sequences contained inside other variables. Furthermore, this example shows how to find all matches in a string rather than stopping at the first match (similar to the g flag in JavaScript's RegEx). var1 := "abc" var2 := 123 MsgBox Deref("%var1%def%var2%") ; Reports abcdef123. Deref(Str) { spo := 1 out := "" while (fpo:=RegexMatch(Str, "(%(.*?)%)|``(.)", &m, spo)) { out .= SubStr(Str, spo, fpo-spo) spo := fpo + StrLen(m[0]) if (m[1]) out .= %m[2]% else switch (m[3]) { case "a": out .= "`a" case "b": out .= "``b" case "f": out .= "`f" case "n": out .= "`n" case "r": out .= "`r" case "t": out .= "`t" case "v": out .= "`v" default: out .= m[3] } } return out SubStr(Str, spo) } RegExReplace - Syntax & Usage | AutoHotkey v2 RegExReplace Replaces occurrences of a pattern (regular expression) inside a string. NewStr := RegExReplace (Haystack, NeedleRegEx , Replacement, &OutputVarCount, Limit, StartingPos) Parameters Haystack Type: String The string whose content is searched and replaced. This may contain binary zero. NeedleRegEx Type: String The pattern to search for, which is a Perl-compatible regular expression (PCRE). The pattern's options (if any) must be included at the beginning of the string followed by a close-parenthesis. For example, the pattern i)abc.*123 would turn on the case-insensitive option and search for "abc", followed by zero or more occurrences of any character, followed by "123". If there are no options, the ")" is optional; for example,)abc is equivalent to abc. Although NeedleRegEx cannot contain binary zero, the pattern \x00 can be used to match a binary zero within Haystack. Replacement Type: String The string to be substituted for each match, which is plain text (not a regular expression). It may include backreferences like $1, which brings in the substring from Haystack that matched the first subpattern. The simplest backreferences are $0 through $9, where $0 is the substring that matched the entire pattern, $1 is the substring that matched the first subpattern, $2 is the second, and so on. For backreferences above 9 (and optionally those below 9), enclose the number in braces; e.g. ${10}, ${11}, and so on. For named subpatterns, enclose the name in braces; e.g. ${SubpatternName}. To specify a literal $, use $$ (this is

the only character that needs such special treatment; backslashes are never needed to escape anything). To convert the case of a subpattern, follow the $ with one of the following characters: U or u (uppercase), L or l (lowercase), T or t (title case, in which the first letter of each word is capitalized but all others are made lowercase). For example, both $U1 and $U{1} transcribe an uppercase version of the first subpattern. Nonexistent backreferences and those that did not match anything in Haystack -- such as one of the subpatterns in (abc)|(xyz) -- are transcribed as empty strings. &OutputVarCount Type: VarRef Specify a reference to a output variable in which to store the number of replacements that occurred (0 if none). Limit Type: Integer If Limit is omitted, it defaults to -1, which replaces all occurrences of the pattern found in Haystack. Otherwise, specify the maximum number of replacements to allow. The part of Haystack to the right of the last replacement is left unchanged. StartingPos Type: Integer If StartingPos is omitted, it defaults to 1 (the beginning of Haystack). Otherwise, specify 2 to start at the second character, 3 to start at the third, and so on. If StartingPos is beyond the length of Haystack, the search starts at the empty string that lies at the end of Haystack (which typically results in no replacements). Specify a negative StartingPos to start at that position from the right. For example, -1 starts at the last character and -2 starts at the next-to-last character. If StartingPos tries to go beyond the left end of Haystack, all of Haystack is searched. Specify 0 to start at the end of Haystack; i.e. the position to the right of the last character. This can be used with zero-width assertions such as (?<=a). Regardless of the value of StartingPos, the return value is always a complete copy of Haystack -- the only difference is that more of its left side might be unaltered compared to what would have happened with a StartingPos of 1. Return Value Type: String This function returns a version of Haystack whose contents have been replaced by the operation. If no replacements are needed, Haystack is returned unaltered. Errors An Error is thrown if: the pattern contains a syntax error; or an error occurred during the execution of the regular expression. For details, see RegExMatch. Options See Options for modifiers such as i)abc, which turns off case-sensitivity in the pattern "abc". Performance To replace simple substrings, use StrReplace because it is faster than RegExReplace. If you know what the maximum number of replacements will be, specifying that for the Limit parameter improves performance because the search can be stopped early (this might also reduce the memory load on the system during the operation). For example, if you know there can be only one match near the beginning of a large string, specify a limit of 1. To improve performance, the 100 most recently used regular expressions are kept cached in memory (in compiled form). The study option (S) can sometimes improve the performance of a regular expression that is used many times (such as in a loop). Remarks Most characters like abc123 can be used literally inside a regular expression. However, the characters \.*?+[{|()^$ must be preceded by a backslash to be seen as literal. For example, \. is a literal period and \\ is a literal backslash. Escaping can be avoided by using \Q...\E. For example: \QLiteral Text\E. Within a regular expression, special characters such as tab and newline can be escaped with either an accent (`) or a backslash (\). For example, `t is the same as \t. To learn the basics of regular expressions (or refresh your memory of pattern syntax), see the RegEx Quick Reference. Related RegExMatch, RegEx Quick Reference, Regular Expression Callouts, StrReplace, InStr Common sources of text data: FileRead, Download, A_Clipboard, GUI Edit controls Examples For general RegEx examples, see the RegEx Quick Reference. Reports "abc123xyz" because the $ allows a match only at the end. MsgBox RegExReplace("abc123123", "123$", "xyz") Reports "123" because a match was achieved via the case-insensitive option. MsgBox RegExReplace("abc123", "i)^ABC") Reports "aaaXYZzzz" by means of the $1 backreference. MsgBox RegExReplace("abcXYZ123", "abc(.*)123", "aaa$1zzz") Reports an empty string and stores 2 in ReplacementCount. MsgBox RegExReplace("abc123abc456", "abc\d+", "", &ReplacementCount) RegRead - Syntax & Usage | AutoHotkey v2 RegRead Reads a value from the registry. Value := RegRead(KeyName, ValueName, Default) Parameters KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a slash, as in this example: \\workstation01\HKEY_LOCAL_MACHINE KeyName can be omitted only if a registry loop is running, in which case it defaults to the key of the current loop item. If the item is a subkey, the full name of that subkey is used by default. If the item is a value, ValueName defaults to the name of that value, but can be overridden. ValueName Type: String The name of the value to retrieve. If blank or omitted, the key's default value will be retrieved (except as noted above). The default value is displayed as "(Default)" by RegEdit. If there is no default value (that is, if RegEdit displays "value not set"), an OSError is thrown. Default Type: Any The value to return if the specified key or value does not exist. If omitted, an OSError is thrown instead of returning a default value. Return Value Type: String or Integer This function returns a value of the specified registry key. Error Handling An OSError is thrown if there was a problem, such as a nonexistent key or value when Default is omitted, or a permission error. A_LastError is set to the result of the operating system's GetLastError() function. Remarks Currently only the following value types are supported: REG_SZ, REG_EXPAND_SZ, REG_MULTI_SZ, REG_DWORD, and REG_BINARY. REG_DWORD values are always expressed as positive decimal numbers. If the number was intended to be negative, convert it to a signed 32-bit integer by using OutputVar := OutputVar << 32 >> 32 or similar. When reading a REG_BINARY key the result is a string of hex characters. For example, the REG_BINARY value of 01,a9,ff,77 will be read as the string 01A9FF77. When reading a REG_MULTI_SZ key, each of the components ends in a linefeed character (`n). If there are no components, an empty string is returned. See FileSelect for an example of how to extract the individual components from the return value. To retrieve and operate upon multiple registry keys or values, consider using a registry-loop. For details about how to access the registry of a remote computer, see the remarks in registry-loop. To read and write entries from the 64-bit sections of the registry in a 32-bit script or vice versa, use SetRegView. Related RegCreateKey, RegDelete, RegDeleteKey, RegWrite, Registry-loop, SetRegView, IniRead Examples Reads a value from the registry and store it in TestValue. TestValue := RegRead ("HKEY_LOCAL_MACHINE\Software\SomeApplication", "TestValue") Retrieves and reports the path of the "Program Files" directory. See EnvGet example #2 for an alternative method. ; The line below ensures that the path of the 64-bit Program Files ; directory is returned if the OS is 64-bit and the script is not. SetRegView 64 ProgramFilesDir := RegRead("HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion", "ProgramFilesDir") MsgBox "Program files are in: " ProgramFilesDir Retrieves the TYPE of a registry value (e.g. REG_SZ or REG_DWORD). MsgBox RegKeyType("HKCU", "Environment", "TEMP") return RegKeyType(RootKey, SubKey, ValueName) ; This function returns the type of the specified value. { Loop Reg, RootKey "\" SubKey if (A_LoopRegName = ValueName) return A_LoopRegType return "Error" } RegWrite - Syntax & Usage | AutoHotkey v2 RegWrite Writes a value to the registry. RegWrite Value, ValueType, KeyName , ValueName RegWrite Value , ValueType, , ValueName Parameters Value Type: String or Integer The value to be written. Long text values can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. ValueType Type: String Must be either REG_SZ, REG_EXPAND_SZ, REG_MULTI_SZ, REG_DWORD, or REG_BINARY. ValueType can be omitted only if KeyName is omitted and the current registry loop item is a value, as noted below. KeyName Type: String The full name of the registry key. This must start with HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT, or HKEY_CURRENT_CONFIG (or the abbreviations for each of these, such as HKLM). To access a remote registry, prepend the computer name and a slash, as in this example: \\workstation01 \HKEY_LOCAL_MACHINE KeyName can be omitted only if a registry loop is running, in which case it defaults to the key of the current loop item. If the item is a subkey, the full name of that subkey is used by default. If the item is a value, ValueType and ValueName default to the type and name of that value, but can be overridden. ValueName Type: String The name of the value that will be written to. If blank or omitted, the key's default value will be used (except as noted above). The default value is displayed as "(Default)" by RegEdit. Error Handling An OSError is thrown on failure. A_LastError is set to the result of the operating system's GetLastError() function. Remarks If KeyName specifies a subkey which does not exist, RegWrite attempts to create it (along with its ancestors, if necessary). Although RegWrite can write directly into a root key, some operating systems might refuse to write into HKEY_CURRENT_USER's top level. To create a key without writing any values to it, use RegCreateKey. If ValueType is REG_DWORD, Value should be between -2147483648 and 4294967295 (0xFFFFFFFF). In the registry, REG_DWORD values are always expressed as positive decimal numbers. To read it as a negative number with means such as RegRead, convert it to a signed 32-bit integer by using OutputVar := OutputVar << 32 >> 32 or similar. When writing a REG_BINARY key, use a string of hex characters, e.g. the REG_BINARY value of 01,a9,ff,77 can be written by using the string 01A9FF77. When writing a REG_MULTI_SZ key, you must separate each component from the next with a linefeed character (`n). The last component may optionally end with a linefeed as well. No blank components are allowed. In other words, do not specify two linefeeds in a row (`n`n) because that will result in a shorter-than-expected value being written to the registry. To retrieve and operate upon multiple registry keys or values, consider using a registry-loop. For details about how to access the registry of a remote computer, see the remarks in registry-loop. To read and write entries from the 64-bit sections of the registry in a 32-bit script or vice versa, use SetRegView. Related RegCreateKey, RegDelete, RegDeleteKey, RegRead, Registry-loop, SetRegView, IniWrite Examples Writes a string to the registry. RegWrite "Test Value", "REG_SZ", "HKEY_LOCAL_MACHINE\SOFTWARE\TestKey", "MyValueName" Writes binary data to the registry. RegWrite "01A9FF77", "REG_BINARY", "HKEY_CURRENT_USER\Software\TEST_APP", "TEST_NAME" Writes a multi-line string to the registry. RegWrite "Line1`nLine2", "REG_MULTI_SZ", "HKEY_CURRENT_USER\Software\TEST_APP", "TEST_NAME" Reload - Syntax & Usage | AutoHotkey v2 Reload Replaces the currently running instance of the script with a new one. Reload This function is useful for scripts that are frequently changed. By assigning a hotkey to this function, you can easily restart the script after saving your changes in an editor. By default, the script can also be reloaded via its tray icon or main window. If the /include switch was passed to the script's current instance, it is automatically passed to the new instance. Any other command-line parameters passed to the script's current instance are not passed to the new instance. To pass such parameters, do not use Reload. Instead, use Run in conjunction with A_AhkPath and A_ScriptFullPath (and A_IsCompiled if the script is ever used in compiled form). Also, include the string /restart as the first parameter (i.e. after the name of the executable), which tells the program to use the same behavior as Reload. See also: command line switches and syntax. When the script restarts, it is launched in its original working directory (the one that was in effect when it was first launched). In other words, SetWorkingDir will not change the working directory that will be used for the new instance. If the script cannot be reloaded -- perhaps because it has a syntax error -- the original instance of the script will continue running. Therefore, the reload function should be followed by whatever actions you want taken in the event of a failure (such as a return to exit the current subroutine). To have the original instance detect the failure, follow this

example: Reload Sleep 1000 ; If successful, the reload will close this instance during the Sleep, so the line below will never be reached. Result := MsgBox("The script could not be reloaded. Would you like to open it for editing?"„ 4) if Result = "Yes" Edit return Previous instances of the script are identified by the same mechanism as for #SingleInstance, with the same limitations. If the script allows multiple instances, Reload may fail to identify the correct instance. The simplest alternative is to Run a new instance and then exit. For example: if A_IsCompiled Run Format('"{1}" /force', A_ScriptFullPath) else Run Format('"{1}" /force "{2}"', A_AhkPath, A_ScriptFullPath) ExitApp Related Edit Examples Press a hotkey to restart the script. ^!r::Reload ; Ctrl+Alt+R Return - Syntax & Usage | AutoHotkey v2 Return Returns from a subroutine to which execution had previously jumped via function-call, Hotkey activation, or other means. Return Expression Parameters Expression This parameter should be omitted except when return is used inside a function. Since this parameter is an expression, all of the following are valid examples: return 3 return "literal string" return MyVar return i + 1 return true ; Returns the number 1 to mean "true". return ItemCount < MaxItems ; Returns a true or false value. return FindColor(TargetColor) Remarks The space or tab after Return is optional if the expression is enclosed in parentheses, as in return(expression). If there is no caller to which to return, Return will do an Exit instead. There are various ways to return multiple values from function to caller described within Returning Values to Caller. Related Functions, Exit, ExitApp Examples The first Return ensures that the subsequent function call is skipped if the preceding condition is true. The second Return is redundant when used at the end of a function like this. #z:: ; Win-Z ^#z:: ; Ctrl-Win-Z { MsgBox "A Win-Z hotkey was pressed." if GetKeyState("Ctrl") return ; Finish early, skipping the function call below. MyFunction() } MyFunction() { Sleep 1000 return ; Redundant when used at the end of the function like this. } Run / RunWait - Syntax & Usage | AutoHotkey v2 Run / RunWait Runs an external program. Unlike Run, RunWait will wait until the program finishes before continuing. Run Target , WorkingDir, Options, &OutputVarPID ExitCode := RunWait(Target , WorkingDir, Options, &OutputVarPID) Parameters Target Type: String A document, URL, executable file (.exe, .com, .bat, etc.), shortcut (.lnk), or system verb to launch (see remarks). If Target is a local file and no path was specified with it, A_WorkingDir will be searched first. If no matching file is found there, the system will search for and launch the file if it is integrated ("known"), e.g. by being contained in one of the PATH folders. To pass parameters, add them immediately after the program or document name. For example, Run 'MyProgram.exe Param1 Param2'. If the program/document name or a parameter contains spaces, it is safest to enclose it in double quotes (even though it may work without them in some cases). For example, Run '"My Program.exe" "param with spaces"'. WorkingDir Type: String The working directory for the launched item. If omitted, the script's own working directory (A_WorkingDir) will be used. Options Type: String If omitted, the function launches Target normally. To change this behavior, specify one or more of the following words: Max: launch maximized Min: launch minimized Hide: launch hidden (cannot be used in combination with either of the above) Note: Some applications (e.g. Calc.exe) do not obey the requested startup state and thus Max/Min/Hide will have no effect. &OutputVarPID Type: VarRef A reference to the output variable in which to store the newly launched program's unique Process ID (PID). The variable will be made blank if the PID could not be determined, which usually happens if a system verb, document, or shortcut is launched rather than a direct executable file. RunWait also supports this parameter, though its OutputVarPID must be checked in another thread (otherwise, the PID will be invalid because the process will have terminated by the time the line following RunWait executes). After the Run function retrieves a PID, any windows to be created by the process might not exist yet. To wait for at least one window to be created, use WinWait "ahk_pid " OutputVarPID. Return Value Type: Integer Unlike Run, RunWait will wait until Target is closed or exits, at which time the return value will be the program's exit code (as a signed 32-bit integer). Some programs will appear to return immediately even though they are still running; these programs spawn another process. Error Handling If Target cannot be launched, an exception is thrown (that is, an error window is displayed) and the current thread is exited, unless the error is caught by a Try/Catch statement. For example: try Run "NonExistingFile" catch MsgBox "File does not exist." The built-in variable A_LastError is set to the result of the operating system's GetLastError() function. Remarks When running a program via ComSpec (cmd.exe) -- perhaps because you need to redirect the program's input or output -- if the path or name of the executable contains spaces, the entire string should be enclosed in an outer pair of quotes. In the following example, the outer quotes are highlighted in yellow: Run A_ComSpec ' /c ""C:\My Utility.exe" "param 1" "second param" >"C:\My File.txt""' Performance may be slightly improved if Target is an exact path, e.g. Run 'C:\Windows\Notepad.exe "C:\My Documents\Test.txt"' rather than Run "C:\My Documents\Test.txt". Special CLSID folders may be opened via Run. For example: Run "::{20d04fe0-3aea-1069-a2d8-08002b30309d}" ; Opens the "My Computer" folder. Run "::{645ff040-5081-101b-9f08-00aa002f954e}" ; Opens the Recycle Bin. System verbs correspond to actions available in a file's right-click menu in the Explorer. If a file is launched without a verb, the default verb (usually "open") for that particular file type will be used. If specified, the verb should be followed by the name of the target file. The following verbs are currently supported: Verb Description *verb Any system-defined or custom verb. For example: Run "*Compile " A_ScriptFullPath. The *RunAs verb may be used in place of the Run as administrator right-click menu item. properties Displays the Explorer's properties window for the indicated file. For example: Run 'properties "C:\My File.txt"' Note: The properties window will automatically close when the script terminates. To prevent this, use WinWait to wait for the window to appear, then use WinWaitClose to wait for the user to close it. find Opens an instance of the Explorer's Search Companion or Find File window at the indicated folder. For example: Run "find D:\" explore Opens an instance of Explorer at the indicated folder. For example: Run "explore " A_ProgramFiles. edit Opens the indicated file for editing. It might not work if the indicated file's type does not have an "edit" action associated with it. For example: Run 'edit "C:\My File.txt"' open Opens the indicated file (normally not needed because it is the default action for most file types). For example: Run 'open "My File.txt"'. print Prints the indicated file with the associated application, if any. For example: Run 'print "My File.txt"' While RunWait is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. Run as Administrator For an executable file, the *RunAs verb is equivalent to selecting Run as administrator from the right-click menu of the file. For example, the following code attempts to restart the current script as admin: full_command_line := DllCall("GetCommandLine", "str") if not (A_IsAdmin or RegExMatch(full_command_line, " /restart(?!\S)")) { try { if A_IsCompiled Run '*RunAs "' A_ScriptFullPath '" /restart' else Run '*RunAs "' A_AhkPath '" /restart "' A_ScriptFullPath '"' } ExitApp } MsgBox "A_IsAdmin: " A_IsAdmin "`nCommand line: " full_command_line If the user cancels the UAC dialog or Run fails for some other reason, the script will simply exit. Using /restart ensures that a single instance prompt is not shown if the new instance of the script starts before ExitApp is called. If UAC is disabled, *RunAs will launch the process without elevating it. Checking for /restart in the command line ensures that the script does not enter a runaway loop in that case. Note that /restart is a built-in switch, so is not included in the array of command-line parameters. The example can be modified to fit the script's needs: If the script absolutely requires admin rights, check A_IsAdmin a second time in case *RunAs failed to elevate the script (i.e. because UAC is disabled). To keep the script running even if the user cancels the UAC prompt, move ExitApp into the try block. To keep the script running even if it failed to restart (i.e. because the script file has been changed or deleted), remove ExitApp and use RunWait instead of Run. On success, /restart causes the new instance to terminate the old one. On failure, the new instance exits and RunWait returns. AutoHotkey's installer registers the RunAs verb for .ahk files, which allows Run "*RunAs script.ahk" to launch a script as admin. Related RunAs, Process functions, Exit, CLSID List, DllCall Examples Run is able to launch Windows system programs from any directory. Note that executable file extensions such as .exe can be omitted. Run "notepad" Run is able to launch URLs: The following opens an internet address in the user's default web browser. Run "https://www.google.com" The following opens the default e-mail application with the recipient filled in. Run "mailto:someone@somedomain.com" The following does the same as above, plus the subject and body. Run "mailto:someone@somedomain.com?subject=This is the subject line&body=This is the message body's text." Opens a document in a maximized application and displays a custom error message on failure. try Run("ReadMe.doc", , "Max") if A_LastError MsgBox "The document could not be launched." Runs the dir command in minimized state and stores the output in a text file. After that, the text file and its properties dialog will be opened. RunWait A_ComSpec " /c dir C:\ >>C:\DirTest.txt", , "Min" Run "C:\DirTest.txt" Run "properties C:\DirTest.txt" Persistent ; Keep the script from exiting, otherwise the properties dialog will close. Run is able to launch CLSIDs: The following opens the recycle bin. Run "::{645ff040-5081-101b-9f08-00aa002f954e}" The following opens the "My Computer" directory. Run "::{20d04fe0-3aea-1069-a2d8-08002b30309d}" To run multiple commands consecutively, use "&&" between each. Run A_ComSpec "/c dir /b > C:\list.txt && type C:\list.txt && pause" The following custom functions can be used to run a command and retrieve its output or to run multiple commands in one go and retrieve their output. For the WshShell object, see Microsoft Docs. MsgBox RunWaitOne("dir " A_ScriptDir) MsgBox RunWaitMany (" (echo Put your commands here, echo each one will be run, echo and you'll get the output.)") RunWaitOne(command) { shell := ComObject("WScript.Shell") ; Execute a single command via cmd.exe exec := shell.Exec(A_ComSpec " /C " command) ; Read and return the command's output return exec.StdOut.ReadAll() } RunWaitMany(commands) { shell := ComObject("WScript.Shell") ; Open cmd.exe with echoing of commands disabled exec := shell.Exec(A_ComSpec " /Q /K echo off") ; Send the commands to execute, separated by newline exec.StdIn.WriteLine(commands "`nexit") ; Always exit at the end! ; Read and return the output of all commands return exec.StdOut.ReadAll() } Executes the given code as a new AutoHotkey process. ExecScript(Script, Wait:=true) { shell := ComObject ("WScript.Shell") exec := shell.Exec("AutoHotkey.exe /ErrorStdOut *") exec.StdIn.Write(script) exec.StdIn.Close() if Wait return exec.StdOut.ReadAll() } ; Example: ib := InputBox("Enter an expression to evaluate as a new script."„„ 'Ord("*")') if ib.result = "Cancel" return result := ExecScript('FileAppend ' ib.value ', "*"') MsgBox "Result: " result RunAs - Syntax & Usage | AutoHotkey v2 RunAs Specifies a set of user credentials to use for all subsequent uses of Run and RunWait. RunAs User, Password, Domain Parameters User Type: String If this and the other parameters are all omitted, the RunAs feature will be turned off, which restores Run and RunWait to their default behavior. Otherwise, this is the username under which new processes will be created. Password Type: String User's password. Domain Type: String User's domain. To use a local account, leave this blank. If that fails to work, try using @YourComputerName. Remarks If the script is running with restricted privileges due to User Account Control (UAC), any programs it launches will typically also be restricted, even if RunAs is used. To elevate a process, use Run *RunAs instead. This function does nothing other than notify AutoHotkey to use (or not use) alternate user credentials for all subsequent uses of Run and RunWait. The credentials are not validated by this function. If an invalid User, Password, or Domain is specified, Run and RunWait will display an error message explaining the problem (unless this error is caught by a Try/Catch statement). While the RunAs feature is in effect, Run and RunWait

will not able to launch documents, URLs, or system verbs. In other words, the file to be launched must be an executable file. The "Secondary Logon" service must be set to manual or automatic for this function to work (the OS should automatically start it upon demand if set to manual). Related Run, RunWait Examples Opens the registry editor as administrator. RunAs "Administrator", "MyPassword" Run "RegEdit.exe" RunAs ; Reset to normal behavior. Send - Syntax & Usage | AutoHotkey v2 Send, SendText, SendInput, SendPlay, SendEvent Sends simulated keystrokes and mouse clicks to the active window. Send Keys SendText Keys SendInput Keys SendPlay Keys SendEvent Keys Parameters Keys Type: String The sequence of keys to send. By default (that is, if neither SendText nor the Raw mode or Text mode is used), the characters ^+!#{} have a special meaning. The characters ^+!# represent the modifier keys Ctrl, Shift, Alt and Win. They affect only the very next key. To send the corresponding modifier key on its own, enclose the key name in braces. To just press (hold down) or release the key, follow the key name with the word "down" or "up" as shown below. Symbol Key Press Release Examples ^ {Ctrl} {Ctrl down} {Ctrl up} Send "^{Home}" presses Ctrl+Home + {Shift} {Shift down} {Shift up} Send "+abC" sends the text "AbC" Send "!+a" presses Alt+Shift+A ! {Alt} {Alt down} {Alt up} Send "!a" presses Alt+A # {LWin} {RWin} {LWin down}{RWin down} {LWin up}{RWin up} Send "#e" holds down Win and then presses E Note: As capital letters are produced by sending Shift, A produces a different effect in some programs than a. For example, !A presses Alt+Shift+A and !a presses Alt+A. If in doubt, use lowercase. The characters {} are used to enclose key names and other options, and to send special characters literally. For example, {Tab} is Tab and {!} is a literal exclamation mark. Enclosing a plain ASCII letter (a-z or A-Z) in braces forces it to be sent as the corresponding virtual keycode, even if the character does not exist on the current keyboard layout. In other words, Send "a" produces the letter "a" while Send "{a}" may or may not produce "a", depending on the keyboard layout. For details, see the remarks below. Send variants Send: By default, Send is synonymous with SendInput; but it can be made a synonym for SendEvent or SendPlay via SendMode. SendText: Similar to Send, except that all characters in Keys are interpreted and sent literally. See Text mode for details. SendInput and SendPlay: SendInput and SendPlay use the same syntax as SendEvent but are generally faster and more reliable. In addition, they buffer any physical keyboard or mouse activity during the send, which prevents the user's keystrokes from being interspersed with those being sent. SendMode can be used to make Send synonymous with SendInput or SendPlay. For more details about each mode, see SendInput and SendPlay below. SendEvent: SendEvent sends keystrokes using the Windows keybd_event function (search Microsoft Docs for details). The rate at which keystrokes are sent is determined by SetKeyDelay. SendMode can be used to make Send synonymous with SendEvent or SendPlay. Special modes The following modes affect the interpretation of the characters in Keys or the behavior of key-sending functions such as Send, SendInput, SendPlay, SendEvent and ControlSend. These modes must be specified as {x} in Keys, where x is either Raw, Text, or Blind. For example, {Raw}. Raw mode The Raw mode can be enabled with {Raw}, which causes all subsequent characters, including the special characters ^+!#{}, to be interpreted literally rather than translating {Enter} to Enter, ^c to Ctrl+C, etc. For example, Send "{Raw}{Tab}" sends {Tab} instead of Tab. The Raw mode does not affect the interpretation of escape sequences and expressions. For example, Send "{Raw}``100`%" sends the string `100%. Text mode The Text mode can be either enabled with {Text}, SendText or ControlSendText, which is similar to the Raw mode, except that no attempt is made to translate characters (other than `r, `n, `t and `b) to keycodes; instead, the fallback method is used for all of the remaining characters. For SendEvent, SendInput and ControlSend, this improves reliability because the characters are much less dependent on correct modifier state. This mode can be combined with the Blind mode to avoid releasing any modifier keys: Send "{Blind}{Text}your text". However, some applications require that the modifier keys be released. `n, `r and `r`n are all translated to a single Enter, unlike the default behavior and Raw mode, which translate `r`n to two Enter. `t is translated to Tab and `b to Backspace, but all other characters are sent without translation. Like the Blind mode, the Text mode ignores SetStoreCapsLockMode (that is, the state of CapsLock is not changed) and does not wait for Win to be released. This is because the Text mode typically does not depend on the state of CapsLock and cannot trigger the system Win+L hotkey. However, this only applies when Keys begins with {Text} or {Blind}{Text}. Blind mode The Blind mode can be enabled with {Blind}, which gives the script more control by disabling a number of things that are normally done automatically to make things work as expected. {Blind} must be the first item in the string to enable the Blind mode. It has the following effects: The Blind mode avoids releasing the modifier keys (Alt, Ctrl, Shift, and Win) if they started out in the down position, unless the modifier is excluded. For example, the hotkey +s::Send "{Blind}abc" would send ABC rather than abc because the user is holding down Shift. Modifier keys are restored differently to allow a Send to turn off a hotkey's modifiers even if the user is still physically holding them down. For example, ^space::Send "{Ctrl up}" automatically pushes Ctrl back down if the user is still physically holding Ctrl, whereas ^space::Send "{Blind}{Ctrl up}" allows Ctrl to be logically up even though it is physically down. SetStoreCapsLockMode is ignored; that is, the state of CapsLock is not changed. Menu masking is disabled. That is, Send omits the extra keystrokes that would otherwise be sent in order to prevent: 1) Start Menu appearance during Win keystrokes (LWin/RWin); 2) menu bar activation during Alt keystrokes. However, the Blind mode does not prevent masking performed by the keyboard hook following activation of a hook hotkey. Send does not wait for Win to be released even if the text contains an L keystroke. This would normally be done to prevent Send from triggering the system "lock workstation" hotkey (Win+L). See Hotkeys for details. The word "Blind" may be followed by one or more modifier symbols (!#^+) to allow those modifiers to be released automatically if needed. For example, *^a::Send "{Blind^}b" would send Shift+B instead of Ctrl+Shift+B if Ctrl+Shift+A was pressed. {Blind!#^+} allows all modifiers to be released if needed, but enables the other effects of the Blind mode. The Blind mode is used internally when remapping a key. For example, the remapping a::b would produce: 1) "b" when you type "a"; 2) uppercase "B" when you type uppercase "A"; and 3) Ctrl+B when you type Ctrl+A. If any modifiers are specified for the source key (including Shift if the source key is an uppercase letter), they are excluded as described above. For example ^a::b produces normal B, not Ctrl+B. {Blind} is not supported by SendText or ControlSendText; use {Blind}{Text} instead. The Blind mode is not completely supported by SendPlay, especially when dealing with the modifier keys (Ctrl, Alt, Shift, and Win). Key names The following table lists the special keys that can be sent (each key name must be enclosed in braces): Key name Description {F1} - {F24} Function keys. For example: {F12} is F12. {!} ! {#} # {+} + {^} ^ {{} { {}} } {Enter} Enter on the main keyboard {Escape} or {Esc} Esc {Space} Space (this is only needed for spaces that appear either at the beginning or the end of the string to be sent -- ones in the middle can be literal spaces) {Tab} Tab {Backspace} or {BS} Backspace {Delete} or {Del} Del {Insert} or {Ins} Ins {Up} ? (up arrow) on main keyboard {Down} ? (down arrow) on main keyboard {Left} ? (left arrow) on main keyboard {Right} ? (right arrow) on main keyboard {Home} Home on main keyboard {End} End on main keyboard {PgUp} PgUp on main keyboard {PgDn} PgDn on main keyboard {CapsLock} CapsLock (using SetCapsLockState is more reliable). Sending {CapsLock} might require SetStoreCapsLockMode False beforehand. {ScrollLock} ScrollLock (see also: SetScrollLockState) {NumLock} NumLock (see also: SetNumLockState) {Control} or {Ctrl} Ctrl (technical info: sends the neutral virtual key but the left scan code) {LControl} or {LCtrl} Left Ctrl (technical info: sends the left virtual key rather than the neutral one) {RControl} or {RCtrl} Right Ctrl {Control down} or {Ctrl down} Holds Ctrl down until {Ctrl up} is sent. To hold down the left or right key instead, replace Ctrl with LCtrl or RCtrl. {Alt} Alt (technical info: sends the neutral virtual key but the left scan code) {LAlt} Left Alt (technical info: sends the left virtual key rather than the neutral one) {RAlt} Right Alt (or AltGr, depending on keyboard layout) {Alt down} Holds Alt down until {Alt up} is sent. To hold down the left or right key instead, replace Alt with LAlt or RAlt. {Shift} Shift (technical info: sends the neutral virtual key but the left scan code) {LShift} Left Shift (technical info: sends the left virtual key rather than the neutral one) {RShift} Right Shift {Shift down} Holds Shift down until {Shift up} is sent. To hold down the left or right key instead, replace Shift with LShift or RShift. {LWin} Left Win {RWin} Right Win {LWin down} Holds the left Win down until {LWin up} is sent {RWin down} Holds the right Win down until {RWin up} is sent {AppsKey} Menu (invokes the right-click or context menu) {Sleep} Sleep. {ASC nnnnn} Sends an Alt+nnnnn keypad combination, which can be used to generate special characters that don't exist on the keyboard. To generate ASCII characters, specify a number between 1 and 255. To generate ANSI characters (standard in most languages), specify a number between 128 and 255, but precede it with a leading zero, e.g. {Asc 0133}. Unicode characters may be generated by specifying a number between 256 and 65535 (without a leading zero). However, this is not supported by all applications. For alternatives, see the section below. {U+nnnn} Sends a Unicode character where nnnn is the hexadecimal value of the character excluding the 0x prefix. This typically isn't needed, because Send and ControlSend automatically support Unicode text. SendInput() or WM_CHAR is used to send the character and the current Send mode has no effect. Characters sent this way usually do not trigger shortcut keys or hotkeys. {vkXX} {scYYY} {vkXXscYYY} Sends a keystroke that has virtual key XX and scan code YYY. For example: Send "{vkFFsc159}". If the sc or vk portion is omitted, the most appropriate value is sent in its place. The values for XX and YYY are hexadecimal and can usually be determined from the main window's View->Key history menu item. See also: Special Keys Warning: Combining vk and sc in this manner is valid only with Send. {Numpad0} - {Numpad9} Numpad digit keys (as seen when NumLock is ON). For example: {Numpad5} is 5. {NumpadDot} . (numpad period) (as seen when NumLock is ON). {NumpadEnter} Enter on keypad {NumpadMult} * (numpad multiplication) {NumpadDiv} / (numpad division) {NumpadAdd} + (numpad addition) {NumpadSub} - (numpad subtraction) {NumpadDel} Del on keypad (this key and the following Numpad keys are used when NumLock is OFF) {NumpadIns} Ins on keypad {NumpadClear} Clear key on keypad (usually 5 when NumLock is OFF). {NumpadUp} ? (up arrow) on keypad {NumpadDown} ? (down arrow) on keypad {NumpadLeft} ? (left arrow) on keypad {NumpadRight} ? (right arrow) on keypad {NumpadHome} Home on keypad {NumpadEnd} End on keypad {NumpadPgUp} PgUp on keypad {NumpadPgDn} PgDn on keypad {Browser_Back} Select the browser "back" button {Browser_Forward} Select the browser "forward" button {Browser_Refresh} Select the browser "refresh" button {Browser_Stop} Select the browser "stop" button {Browser_Search} Select the browser "search" button {Browser_Favorites} Select the browser "favorites" button {Browser_Home} Launch the browser and go to the home page {Volume_Mute} Mute/unmute the master volume. Usually equivalent to SoundSetMute -1. {Volume_Down} Reduce the master volume. Usually equivalent to SoundSetVolume -5. {Volume_Up} Increase the master volume. Usually equivalent to SoundSetVolume "+5". {Media_Next} Select next track in media player {Media_Prev} Select previous track in media player {Media_Stop} Stop media player {Media_Play_Pause} Play/pause media player {Launch_Mail} Launch the email application {Launch_Media} Launch media player {Launch_App1} Launch user app1 {Launch_App2} Launch user app2 {PrintScreen} PrtSc {CtrlBreak} Ctrl+Pause {Pause} Pause {Click [Options]} Sends a mouse click using the same options available in the Click function. For example, Send "{Click}" would click the left mouse button once at the mouse cursor's current position, and Send "{Click 100

200}" would click at coordinates 100, 200 (based on CoordMode). To move the mouse without clicking, specify 0 after the coordinates; for example: Send "{Click 100 200 0}". The delay between mouse clicks is determined by SetMouseDelay (not SetKeyDelay). {WheelDown}, {WheelUp}, {WheelLeft}, {WheelRight}, {LButton}, {RButton}, {MButton}, {XButton1}, {XButton2} Sends a mouse button event at the cursor's current position (to have control over position and other options, use {Click} above). The delay between mouse clicks is determined by SetMouseDelay. LButton and RButton correspond to the primary and secondary mouse buttons. Normally the primary mouse button (LButton) is on the left, but the user may swap the buttons via system settings. {Blind} Enables the Blind mode, which gives the script more control by disabling a number of things that are normally done automatically to make things generally work as expected. The string {Blind} must occur at the beginning of the string. {Raw} Enables the Raw mode, which causes the following characters to be interpreted literally: ^+!#{}. Although the string {Raw} need not occur at the beginning of the string, once specified, it stays in effect for the remainder of the string. {Text} Enables the Text mode, which sends a stream of characters rather than keystrokes. Like the Raw mode, the Text mode causes the following characters to be interpreted literally: ^+!#{}. Although the string {Text} need not occur at the beginning of the string, once specified, it stays in effect for the remainder of the string. Repeating or Holding Down a Key To repeat a keystroke: Enclose in braces the name of the key followed by the number of times to repeat it. For example: Send "{DEL 4}" ; Presses the Delete key 4 times. Send "{S 30}" ; Sends 30 uppercase S characters. Send "+{TAB 4}" ; Presses Shift-Tab 4 times. To hold down or release a key: Enclose in braces the name of the key followed by the word Down or Up. For example: Send "{b down}{b up}" Send "{TAB down}{TAB up}" Send "{Up down}" ; Press down the up-arrow key. Sleep 1000 ; Keep it down for one second. Send "{Up up}" ; Release the up-arrow key. When a key is held down via the method above, it does not begin auto-repeating like it would if you were physically holding it down (this is because auto-repeat is a driver/hardware feature). However, a Loop can be used to simulate auto-repeat. The following example sends 20 tab keystrokes: Loop 20 { Send "{Tab down}" ; Auto-repeat consists of consecutive down-events (with no up-events). Sleep 30 ; The number of milliseconds between keystrokes (or use SetKeyDelay). } Send "{Tab up}" ; Release the key. By default, Send will not automatically release a modifier key (Control, Shift, Alt, and Win) if that modifier key was "pressed down" by sending it. For example, Send "a" may behave similar to Send "{Blind}{Ctrl up}a{Ctrl down}" if the user is physically holding Ctrl, but Send "{Ctrl Down}" followed by Send "a" will produce Ctrl+A. DownTemp and DownR can be used to override this behavior. DownTemp and DownR have the same effect as Down except for the modifier keys (Control, Shift, Alt, and Win). DownTemp tells subsequent sends that the key is not permanently down, and may be released whenever a keystroke calls for it. For example, Send "{Control DownTemp}" followed later by Send "a" would produce A, not Ctrl+A. Any use of Send may potentially release the modifier permanently, so DownTemp is not ideal for remapping modifier keys. DownR (where "R" stands for remapping, which is its main use) tells subsequent sends that if the key is automatically released, it should be pressed down again when send is finished. For example, Send "{Control DownR}" followed later by Send "a" would produce A, not Ctrl+A, but will leave Ctrl in the pressed state for use with keyboard shortcuts. In other words, DownR has an effect similar to physically pressing the key. If a character does not correspond to a virtual key on the current keyboard layout, it cannot be "pressed" or "released". For example, Send "{Âµ up}" has no effect on most layouts, and Send "{Âµ down}" is equivalent to Send "Âµ". General Remarks Characters vs. keys: By default, characters are sent by first translating them to keystrokes. If this translation is not possible (that is, if the current keyboard layout does not contain a key or key combination which produces that character), the character is sent by one of following fallback methods: SendEvent and SendInput use SendInput() with the KEYEVENTF_UNICODE flag. SendPlay uses the Alt+nnnnn method, which produces Unicode only if supported by the target application. ControlSend posts a WM_CHAR message. Note: Characters sent using any of the above methods usually do not trigger keyboard shortcuts or hotkeys. For characters in the range a-z or A-Z (plain ASCII letters), each character which does not exist in the current keyboard layout may be sent either as a character or as the corresponding virtual keycode (vk41-vk5A): If a naked letter is sent (that is, without modifiers or braces), or if Raw mode is in effect, it is sent as a character. For example, Send "{Raw}Regards" sends the expected text, even though pressing R (vk52) produces some other character (such as ?š on the Russian layout). {Raw} can be omitted in this case, unless a modifier key was put into effect by a prior Send. If one or more modifier keys have been put into effect by the Send function, or if the letter is wrapped in braces, it is sent as a keycode (modified with Shift if the letter is upper-case). This allows the script to easily activate standard keyboard shortcuts. For example, ^c and {Ctrl down}c{Ctrl up} activate the standard Ctrl+C shortcut and {c} is equivalent to {vk43}. If the letter exists in the current keyboard layout, it is always sent as whichever keycode the layout associates with that letter (unless the Text mode is used, in which case the character is sent by other means). In other words, the section above is only relevant for non-Latin based layouts such as Russian. Modifier State: When Send is required to change the state of the Win or Alt modifier keys (such as if the user was holding one of those keys), it may inject additional keystrokes (Ctrl by default) to prevent the Start menu or window menu from appearing. For details, see A_MenuMaskKey. BlockInput Compared to SendInput/SendPlay: Although the BlockInput function can be used to prevent any keystrokes physically typed by the user from disrupting the flow of simulated keystrokes, it is often better to use SendInput or SendPlay so that keystrokes and mouse clicks become uninterruptible. This is because unlike BlockInput, SendInput/Play does not discard what the user types during the send; instead, such keystrokes are buffered and sent afterward. When sending a large number of keystrokes, a continuation section can be used to improve readability and maintainability. Since the operating system does not allow simulation of the Ctrl+Alt+Del combination, doing something like Send "^!{Delete}" will have no effect. Send may have no effect if the active window is running with administrative privileges and the script is not. This is due to a security mechanism called User Interface Privilege Isolation. SendInput SendInput is generally the preferred method to send keystrokes and mouse clicks because of its superior speed and reliability. Under most conditions, SendInput is nearly instantaneous, even when sending long strings. Since SendInput is so fast, it is also more reliable because there is less opportunity for some other window to pop up unexpectedly and intercept the keystrokes. Reliability is further improved by the fact that anything the user types during a SendInput is postponed until afterward. Unlike the other sending modes, the operating system limits SendInput to about 5000 characters (this may vary depending on the operating system's version and performance settings). Characters and events beyond this limit are not sent. Note: SendInput ignores SetKeyDelay because the operating system does not support a delay in this mode. However, when SendInput reverts to SendEvent under the conditions described below, it uses SetKeyDelay -1, 0 (unless SendEvent's KeyDelay is -1,-1, in which case -1,-1 is used). When SendInput reverts to SendPlay, it uses SendPlay's KeyDelay. If a script other than the one executing SendInput has a low-level keyboard hook installed, SendInput automatically reverts to SendEvent (or SendPlay if SendMode "InputThenPlay" is in effect). This is done because the presence of an external hook disables all of SendInput's advantages, making it inferior to both SendPlay and SendEvent. However, since SendInput is unable to detect a low-level hook in programs other than AutoHotkey v1.0.43+, it will not revert in these cases, making it less reliable than SendPlay/Event. When SendInput sends mouse clicks by means such as {Click}, and CoordMode "Mouse", "Window" or CoordMode "Mouse", "Client" is in effect, every click will be relative to the window that was active at the start of the send. Therefore, if SendInput intentionally activates another window (by means such as alt-tab), the coordinates of subsequent clicks within the same function will be wrong if they were intended to be relative to the new window rather than the old one. SendPlay Warning: SendPlay may have no effect at all if UAC is enabled, even if the script is running as an administrator. For more information, refer to the FAQ. SendPlay's biggest advantage is its ability to "play back" keystrokes and mouse clicks in a broader variety of games than the other modes. For example, a particular game may accept hotstrings only when they have the SendPlay option. Of the three sending modes, SendPlay is the most unusual because it does not simulate keystrokes and mouse clicks per se. Instead, it creates a series of events (messages) that flow directly to the active window (similar to ControlSend, but at a lower level). Consequently, SendPlay does not trigger hotkeys or hotstrings. Like SendInput, SendPlay's keystrokes do not get interspersed with keystrokes typed by the user. Thus, if the user happens to type something during a SendPlay, those keystrokes are postponed until afterward. Although SendPlay is considerably slower than SendInput, it is usually faster than the traditional SendEvent mode (even when KeyDelay is -1). Both Win (LWin and RWin) are automatically blocked during a SendPlay if the keyboard hook is installed. This prevents the Start Menu from appearing if the user accidentally presses Win during the send. By contrast, keys other than LWin and RWin do not need to be blocked because the operating system automatically postpones them until after the SendPlay (via buffering). SendPlay does not use the standard settings of SetKeyDelay and SetMouseDelay. Instead, it defaults to no delay at all, which can be changed as shown in the following examples: SetKeyDelay 0, 10, "Play" ; Note that both 0 and -1 are the same in SendPlay mode. SetMouseDelay 10, "Play" SendPlay is unable to turn on or off CapsLock, NumLock, or ScrollLock. Similarly, it is unable to change a key's state as seen by GetKeyState unless the keystrokes are sent to one of the script's own windows. Even then, any changes to the left/right modifier keys (e.g. RControl) can be detected only via their neutral counterparts (e.g. Control). Also, SendPlay has other limitations described on the SendMode page. Unlike SendInput and SendEvent, the user may interrupt a SendPlay by pressing Ctrl+Alt+Del or Ctrl+Esc. When this happens, the remaining keystrokes are not sent but the script continues executing as though the SendPlay had completed normally. Although SendPlay can send LWin and RWin events, they are sent directly to the active window rather than performing their native operating system function. To work around this, use SendEvent. For example, SendEvent "#r" would show the Start Menu's Run dialog. Related SendMode, SetKeyDelay, SetStoreCapsLockMode, Escape sequences (e.g. `n), ControlSend, BlockInput, Hotstrings, WinActivate Examples Types a two-line signature. Send "Sincerely,{enter}John Smith" Selects the File->Save menu (Alt+F followed by S). Send "!fs" Jumps to the end of the text then send four shift+left-arrow keystrokes. Send "{End}+{Left 4}" Sends a long series of raw characters via the fastest method. SendInput "{Raw}A long series of raw characters sent via the fastest method." SendLevel - Syntax & Usage | AutoHotkey v2 SendLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. SendLevel Level Parameters Level Type: Integer An integer between 0 and 100. Return Value Type: Integer This function returns the previous setting. General Remarks By default, hook hotkeys and hotstrings ignore keyboard and mouse events generated by any AutoHotkey script. In some cases it can be useful to override this behaviour; for instance, to allow a remapped key to be used to trigger other hotkeys. SendLevel and #InputLevel provide the means to achieve this. SendLevel sets the level for events generated by the current script thread, while #InputLevel sets the level for any hotkeys or hotstrings beneath it. For any event generated by a script to trigger a hook hotkey or hotstring, the send level of the event must be higher than the input level of the hotkey or hotstring. Compatibility: SendPlay is not affected by SendLevel. SendInput is affected by SendLevel, but the script's own hook

hotkeys cannot be activated while a SendInput is in progress, since it temporarily deactivates the hook. However, when Send or SendInput reverts to SendEvent, it is able to activate the script's own hotkeys. Hotkeys using the "reg" method are incapable of distinguishing physical and artificial input, so are not affected by SendLevel. However, hotkeys above level 0 always use the keyboard or mouse hook. Hotstrings use #InputLevel only to determine whether the last typed character should trigger a hotstring. For instance, the hotstring ::btw:: can be triggered regardless of #InputLevel by sending btw at level 1 or higher and physically typing an ending character. This is because hotstring recognition works by collecting input from all levels except level 0 into a single global buffer. Auto-replace hotstrings always generate keystrokes at level 0, since it is usually undesirable for the replacement text to trigger another hotstring or hotkey. To work around this, use a non-auto-replace hotstring and the SendEvent function. Characters sent by the ASC (Alt+nnnnn) method cannot trigger a hotstring, even if SendLevel is used. Characters sent by SendEvent with the {Text} mode, {U+nnnn} or Unicode fallback method can trigger hotstrings. The built-in variable A_SendLevel contains the current setting. Every newly launched hotkey or hotstring thread starts off with a send level equal to the input level of the hotkey or hotstring. Every other newly launched thread (such as a custom menu item or timed subroutine) starts off fresh with the default setting, which is typically 0 but may be changed by using this function during script startup. If SendLevel is used during script startup, it also affects keyboard and mouse remapping. AutoHotkey versions older than v1.1.06 behave as though #InputLevel 0 and SendLevel 0 are in effect. Related #InputLevel, Send, Click, MouseClick, MouseClickDrag Examples SendLevel allows to trigger hotkeys and hotstrings of another script, which normally would not be the case. SendLevel 1 SendEvent "btw{Space}" ; Produces "by the way ". ; This may be defined in a separate script: ::btw::by the way SendMessage - Syntax & Usage | AutoHotkey v2 SendMessage Sends a message to a window or control and waits for acknowledgement. Result := SendMessage(Msg , wParam, lParam, Control, WinTitle, WinText, ExcludeTitle, ExcludeText, Timeout) Parameters Msg Type: Integer The message number to send. See the message list to determine the number. wParam, lParam Type: Integer or Object The message parameters. If omitted, each parameter defaults to 0. Each parameter must be an integer or an object with a Ptr property, such as a Buffer. For messages which require a pointer to a string, use a Buffer or the StrPtr function. If the string contained by a variable is changed by passing the variable's address to SendMessage, the variable's length must be updated afterward by calling VarSetStrCapacity(&MyVar, -1). If AutoHotkey or the target window is 32-bit, only the parameter's low 32 bits are used; that is, values are truncated if outside the range -2147483648 to 2147483647 (-0x80000000 to 0x7FFFFFFF) for signed values, or 0 to 4294967295 (0xFFFFFFFF) for unsigned values. If AutoHotkey and the target window are both 64-bit, any integer value supported by AutoHotkey can be used. Control Type: String, Integer or Object If this parameter is omitted, the message will be sent directly to the target window rather than one of its controls. Otherwise, this parameter can be the control's ClassNN, text or HWND, or an object with a Hwnd property. For details, see The Control Parameter. If this parameter specifies a HWND (as an integer or object), it is not required to be the HWND of a control (child window). That is, it can also be the HWND of a top-level window. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Timeout Type: Integer The maximum number of milliseconds to wait for the target window to process the message. If omitted, it defaults to 5000 (milliseconds). If the message is not processed within this time, a TimeoutError is thrown. Return Value Type: Integer This function returns the result of the message, which might sometimes be a "reply" depending on the nature of the message and its target window. The range of possible values depends on the target window and the version of AutoHotkey that is running. When using a 32-bit version of AutoHotkey, or if the target window is 32-bit, the result is a 32-bit unsigned integer between 0 and 4294967295. When using the 64-bit version of AutoHotkey with a 64-bit window, the result is a 64-bit signed integer between -9223372036854775808 and 9223372036854775807. If the result is intended to be a 32-bit signed integer (a value from -2147483648 to 2147483648), it can be truncated to 32-bit and converted to a signed value as follows: MsgReply := MsgReply << 32 >> 32 This conversion may be necessary even on AutoHotkey 64-bit, because results from 32-bit windows are zero-extended. For example, a result of -1 from a 32-bit window is seen as 0xFFFFFFFF on any version of AutoHotkey, whereas a result of -1 from a 64-bit window is seen as 0xFFFFFFFF on AutoHotkey 32-bit and -1 on AutoHotkey 64-bit. Error Handling A TargetError is thrown if the window or control could not be found. A TimeoutError is thrown if the message timed out. An OSError is thrown if a message could not be sent. For example, if the target window is running at a higher integrity level than the script (i.e. it is running as admin while the script is not), messages may be blocked. Remarks This function should be used with caution because sending a message to the wrong window (or sending an invalid message) might cause unexpected behavior or even crash the target application. This is because most applications are not designed to expect certain types of messages from external sources. SendMessage waits for the target window to process the message, up until the timeout period expires. By contrast, PostMessage places the message in the message queue associated with the target window without waiting for acknowledgement or reply. String parameters must be passed by address. For example: ListVars WinWaitActive "ahk_class AutoHotkey" SendMessage 0x000C, 0, StrPtr("New Title") ; 0X000C is WM_SETTEXT To send a message to all windows in the system, including those that are hidden or disabled, specify 0xFFFF for WinTitle (0xFFFF is HWND_BROADCAST). This technique should be used only for messages intended to be broadcast, such as the following example: SendMessage 0x001A,,,, 0xFFFF ; 0x001A is WM_SETTINGCHANGE To have a script receive a message, use OnMessage. See the Message Tutorial for an introduction to using this function. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related PostMessage, Message List, Message Tutorial, OnMessage, Automating Winamp, DllCall, ControlSend, MenuSelect Examples Turns off the monitor via hotkey. In the SendMessage line, replace the number 2 with -1 to turn the monitor on or replace it with 1 to activate the monitor's low-power mode. #o:: ; Win+O hotkey { Sleep 1000 ; Give user a chance to release keys (in case their release would wake up the monitor again). ; Turn Monitor Off: SendMessage 0x0112, 0xF170, 2,, "Program Manager" ; 0x0112 is WM_SYSCOMMAND, 0xF170 is SC_MONITORPOWER. } Starts the user's chosen screen saver. SendMessage 0x0112, 0xF140, 0,, "Program Manager" ; 0x0112 is WM_SYSCOMMAND, and 0xF140 is SC_SCREENSAVE. Scrolls up by one line (for a control that has a vertical scroll bar). SendMessage 0x0115, 0, 0, ControlGetFocus("A") Scrolls down by one line (for a control that has a vertical scroll bar). SendMessage 0x0115, 1, 0, ControlGetFocus("A") Asks Winamp which track number is currently active (see Automating Winamp for more information). SetTitleMatchMode 2 track := SendMessage(0x0400, 0, 120,, "- Winamp") track++ ; Winamp's count starts at 0, so adjust by 1. MsgBox "Track #" track " is active or playing." Finds the process ID of an AHK script (an alternative to WinGetPID). SetTitleMatchMode 2 DetectHiddenWindows true pid := SendMessage(0x0044, 0x405, 0, , "SomeOtherScript.ahk - AutoHotkey v") MsgBox pid " is the process id." SendMode - Syntax & Usage | AutoHotkey v2 SendMode Makes Send synonymous with SendEvent or SendPlay rather than the default (SendInput). Also makes Click and MouseMove/Click/Drag use the specified method. SendMode Mode Parameters Mode Type: String Specify one of the following words: Event: Switches to the SendEvent method for Send, SendText, Click, and MouseMove/Click/Drag. Input: This is the starting default used by all scripts. It uses the SendInput method for Send, SendText, Click, and MouseMove/Click/Drag. Known limitations: Windows Explorer ignores SendInput's simulation of certain navigational hotkeys such as Alt+?. To work around this, use either SendEvent "!{Left}" or SendInput "{Backspace}". InputThenPlay: Same as above except that rather than falling back to Event mode when SendInput is unavailable, it reverts to Play mode (below). This also causes the SendInput function itself to revert to Play mode when SendInput is unavailable. Play: Switches to the SendPlay method for Send, SendText, Click, and MouseMove/Click/Drag. Known limitations: Characters that do not exist in the current keyboard layout (such as Ô in English) cannot be sent. To work around this, use SendEvent. Simulated mouse dragging might have no effect in RichEdit controls (and possibly others) such as those of WordPad and Metapad. To use an alternate mode for a particular drag, follow this example: SendEvent "{Click 6 52 Down}{Click 45 52 Up}". Simulated mouse wheel rotation produces movement in only one direction (usually downward, but upward in some applications). Also, wheel rotation might have no effect in applications such as MS Word and Notepad. To use an alternate mode for a particular rotation, follow this example: SendEvent "{WheelDown 5}". If SendMode "Play" is called during script startup, all remapped keys are affected and might lose some of their functionality. See SendPlay remapping limitations for details. SendPlay does not trigger AutoHotkey's hotkeys or hotstrings, or global hotkeys registered by other programs or the OS. Return Value Type: String This function returns the previous setting. Remarks Since SendMode also changes the mode of Click and MouseMove/Click/Drag, there may be times when you wish to use a different mode for a particular mouse event. The easiest way to do this is via {Click}. For example: SendEvent "{Click 100 200}" ; SendEvent uses the older, traditional method of clicking. If SendMode is used during script startup, it also affects keyboard and mouse remapping. In particular, if you use SendMode "Play" with remapping, see SendPlay remapping limitations. The built-in variable A_SendMode contains the current setting. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related Send, SetKeyDelay, SetMouseDelay, Click, MouseClick, MouseClickDrag, MouseMove Examples Makes Send synonymous with SendInput. Recommended for new scripts due to its superior speed and reliability. SendMode "Input" Makes Send synonymous with SendInput, but falls back to SendPlay if SendInput is not available. SendMode "InputThenPlay" SetControlDelay - Syntax & Usage | AutoHotkey v2 SetControlDelay Sets the delay that will occur after each control-modifying function. SetControlDelay Delay Parameters Delay Type: Integer Time in milliseconds. Use -1 for no delay at all and 0 for the smallest possible delay. If unset, the default delay is 20. Return Value Type: Integer This function returns the previous setting. Remarks A short delay (sleep) is done automatically after every Control function that changes a control. This is done to improve the reliability of scripts because a control sometimes needs a period of "rest" after being changed by one of these functions. The rest period allows it to update itself and respond to the next function that the script may attempt to send to it. Specifically, SetControlDelay affects the following functions: ControlAddItem, ControlChooseIndex, ControlChooseString, ControlClick, ControlDeleteItem, EditPaste, ControlFindItem, ControlFocus, ControlHide, ControlHideDropDown, ControlMove, ControlSetChecked, ControlSetEnabled, ControlSetText, ControlShow, ControlShowDropDown. ControlSend is not affected; it uses SetKeyDelay. Although a delay of -1 (no delay at all) is allowed, it is recommended that at least 0 be used, to increase confidence that the script will run correctly even when the CPU is under load. A delay of 0 internally executes a Sleep(0), which yields

the remainder of the script's timeslice to any other process that may need it. If there is none, Sleep(0) will not sleep at all. If the CPU is slow or under load, or if window animation is enabled, higher delay values may be needed. The built-in variable A_ControlDelay contains the current setting and can also be assigned a new value instead of calling SetControlDelay. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related Control functions, SetWinDelay, SetKeyDelay, SetMouseDelay Examples Causes the smallest possible delay to occur after each control-modifying function. SetControlDelay 0 SetDefaultMouseSpeed - Syntax & Usage | AutoHotkey v2 SetDefaultMouseSpeed Sets the mouse speed that will be used if unspecified in Click and MouseMove/Click/Drag. SetDefaultMouseSpeed Speed Parameters Speed Type: Integer The speed to move the mouse in the range 0 (fastest) to 100 (slowest). Note: A speed of 0 will move the mouse instantly. Return Value Type: Integer This function returns the previous setting. Remarks SetDefaultMouseSpeed is ignored for SendInput/Play modes; they move the mouse instantaneously (except when SendInput reverts to SendEvent; also, SetMouseDelay has a mode that applies to SendPlay). To visually move the mouse more slowly -- such as a script that performs a demonstration for an audience -- use SendEvent "{Click 100 200}" or SendMode "Event" (optionally in conjuction with BlockInput). If this function is not used, the default mouse speed is 2. The built-in variable A_DefaultMouseSpeed contains the current setting. The functions MouseClick, MouseMove, and MouseClickDrag all have a parameter to override the default mouse speed. Whenever Speed is greater than zero, SetMouseDelay also influences the speed by producing a delay after each incremental move the mouse makes toward its destination. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related SetMouseDelay, SendMode, Click, MouseClick, MouseMove, MouseClickDrag, SetWinDelay, SetControlDelay, SetKeyDelay, SetKeyDelay Examples Causes the mouse cursor to be moved instantly. SetDefaultMouseSpeed 0 SetKeyDelay - Syntax & Usage | AutoHotkey v2 SetKeyDelay Sets the delay that will occur after each keystroke sent by Send or ControlSend. SetKeyDelay Delay, PressDuration, "Play" Parameters Delay Type: Integer Time in milliseconds. Use -1 for no delay at all and 0 for the smallest possible delay (however, if the Play parameter is present, both 0 and -1 produce no delay). Leave this parameter blank to retain the current Delay. If SetKeyDelay is never used by a script, the default Delay for the tradional SendEvent mode is 10. For SendPlay mode, the default Delay is -1. The default PressDuration (below) is -1 for both modes. PressDuration Type: Integer Certain games and other specialized applications may require a delay inside each keystroke; that is, after the press of the key but before its release. Use -1 for no delay at all (default) and 0 for the smallest possible delay (however, if the Play parameter is present, both 0 and -1 produce no delay). Omit this parameter to leave the current PressDuration unchanged. Note: PressDuration also produces a delay after any change to the modifier key state (Ctrl, Alt, Shift, and Win) needed to support the keys being sent. Play Type: String The word Play applies the above settings to the SendPlay mode rather than the traditional SendEvent mode. If a script never uses this parameter, the delay is always -1/-1 for SendPlay. Remarks Note: SetKeyDelay is not obeyed by SendInput; there is no delay between keystrokes in that mode. This same is true for Send when SendMode Input is in effect. A short delay (sleep) is done automatically after every keystroke sent by Send or ControlSend. This is done to improve the reliability of scripts because a window sometimes can't keep up with a rapid flood of keystrokes. During the delay (sleep), the current thread is made uninterruptible. Due to the granularity of the OS's time-keeping system, delays might be rounded up to the nearest multiple of 10 or 15. For Send/SendEvent mode, a delay of 0 internally executes a Sleep(0), which yields the remainder of the script's timeslice to any other process that may need it. If there is none, Sleep(0) will not sleep at all. By contrast, a delay of -1 will never sleep. For better reliability, 0 is recommended as an alternative to -1. When the delay is set to -1, a script's process-priority becomes an important factor in how fast it can send keystrokes when using the traditional SendEvent mode. To raise a script's priority, use ProcessSetPriority "High". Although this typically causes keystrokes to be sent faster than the active window can process them, the system automatically buffers them. Buffered keystrokes continue to arrive in the target window after the Send function completes (even if the window is no longer active). This is usually harmless because any subsequent keystrokes sent to the same window get queued up behind the ones already in the buffer. The built-in variable A_KeyDelay contains the current setting of Delay for Send/SendEvent mode. A_KeyDuration contains the setting for PressDuration, while A_KeyDelayPlay and A_KeyDurationPlay contain the settings for SendPlay. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related Send, ControlSend, SendMode, SetMouseDelay, SetControlDelay, SetWinDelay, Click Examples Causes the smallest possible delay to occur after each keystroke sent via Send or ControlSend. SetKeyDelay 0 SetMouseDelay - Syntax & Usage | AutoHotkey v2 SetMouseDelay Sets the delay that will occur after each mouse movement or click. SetMouseDelay Delay , "Play" Parameters Delay Type: Integer Time in milliseconds. Use -1 for no delay at all and 0 for the smallest possible delay (however, if the Play parameter is present, both 0 and -1 produce no delay). If unset, the default delay is 10 for the traditional SendEvent mode and -1 for SendPlay mode. Play Type: String The word Play applies the delay to the SendPlay mode rather than the traditional Send/SendEvent mode. If a script never uses this parameter, the delay is always -1 for SendPlay. Return Value Type: Integer This function returns the previous setting. Remarks A short delay (sleep) is done automatically after every mouse movement or click generated by Click and MouseMove/Click/Drag (except for SendInput mode). This is done to improve the reliability of scripts because a window sometimes can't keep up with a rapid flood of mouse events. Due to the granularity of the OS's time-keeping system, delays might be rounded up to the nearest multiple of 10 or 15. A delay of 0 internally executes a Sleep(0), which yields the remainder of the script's timeslice to any other process that may need it. If there is none, Sleep(0) will not sleep at all. By contrast, a delay of -1 will never sleep. The built-in variable A_MouseDelay contains the current setting for Send/SendEvent mode. A_MouseDelayPlay contains the current setting for SendPlay mode. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related SetDefaultMouseSpeed, Click, MouseMove, MouseClick, MouseClickDrag, SendMode, SetKeyDelay, SetControlDelay, SetWinDelay Examples Causes the smallest possible delay to occur after each mouse movement or click. SetMouseDelay 0 SetCapsLockState / SetNumLockState / SetScrollLockState - Syntax & Usage | AutoHotkey v2 SetCapsLockState / SetNumLockState / SetScrollLockState Sets the state of CapsLock/NumLock/ScrollLock. Can also force the key to stay on or off. SetCapsLockState State SetNumLockState State SetScrollLockState State Parameters State Type: String or Integer If this parameter is omitted, the AlwaysOn/Off attribute of the key is removed (if present). Otherwise, specify one of the following words: On or 1 (true): Turns on the key and removes the AlwaysOn/Off attribute of the key (if present). Off or 0 (false): Turns off the key and removes the AlwaysOn/Off attribute of the key (if present). AlwaysOn: Forces the key to stay on permanently. AlwaysOff: Forces the key to stay off permanently. Remarks Alternatively to example #3 below, a key can also be toggled to its opposite state via the Send function; for example: Send "{CapsLock}". However, sending {CapsLock} might require SetStoreCapsLockMode False beforehand. Keeping a key AlwaysOn or AlwaysOff requires the keyboard hook, which will be automatically installed in such cases. Related SetStoreCapsLockMode, GetKeyState Examples Turns on NumLock and removes the AlwaysOn/Off attribute of the key (if present). SetNumLockState True Forces ScrollLock to stay off permanently. SetScrollLockState "AlwaysOff" Toggles CapsLock to its opposite state. SetCapsLockState !GetKeyState("CapsLock", "T") SetRegView - Syntax & Usage | AutoHotkey v2 SetRegView Sets the registry view used by RegRead, RegWrite, RegDelete, RegDeleteKey and Loop Reg, allowing them in a 32-bit script to access the 64-bit registry view and vice versa. SetRegView RegView Parameters RegView Type: Integer or String Specify 32 to view the registry as a 32-bit application would, or 64 to view the registry as a 64-bit application would. Specify the word Default to restore normal behaviour. Return Value Type: String This function returns the previous setting. Remarks This function is only useful on Windows 64-bit. It has no effect on Windows 32-bit. On 64-bit systems, 32-bit applications run on a subsystem of Windows called WOW64. By default, the system redirects certain registry keys to prevent conflicts. For example, in a 32-bit script, HKLM\SOFTWARE\AutoHotkey is redirected to HKLM\SOFTWARE\Wow6432Node\AutoHotkey. SetRegView allows the registry functions in a 32-bit script to access redirected keys in the 64-bit registry view and vice versa. The built-in variable A_RegView contains the current setting. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related RegRead, RegWrite, RegCreateKey, RegDelete, RegDeleteKey, Loop Reg Examples Shows how to set a specific registry view, and how registry redirection affects the script. ; Access the registry as a 32-bit application would. SetRegView 32 RegWrite "REG_SZ", "HKLM\SOFTWARE\Test.ahk", "Value", 123 ; Access the registry as a 64-bit application would. SetRegView 64 value := RegRead("HKLM\SOFTWARE\Wow6432Node\Test.ahk", "Value") RegDelete "HKLM\SOFTWARE\Wow6432Node\Test.ahk" MsgBox "Read value '" value "' via Wow6432Node." ; Restore the registry view to the default, which ; depends on whether the script is 32-bit or 64-bit. SetRegView "Default" ;... Shows how to detect the type of EXE and operating system on which the script is running. if (A_PtrSize = 8) script_is := "64-bit" else ; if (A_PtrSize = 4) script_is := "32-bit" if (A_Is64bitOS) OS_is := "64-bit" else OS_is := "32-bit, which has only a single registry view" MsgBox "This script is " script_is ", and the OS is " OS_is "." SetStoreCapsLockMode - Syntax & Usage | AutoHotkey v2 SetStoreCapsLockMode Whether to restore the state of CapsLock after a Send. SetStoreCapsLockMode Mode Parameters Mode Type: Boolean One of the following values: 1 or True: This is the initial setting for all scripts: CapsLock will be restored to its former value if Send needed to change it temporarily for its operation. 0 or False: The state of CapsLock is not changed at all. As a result, Send will invert the case of the characters if CapsLock happens to be ON during the operation. Return Value Type: Boolean This function returns the previous setting. Remarks This means that CapsLock will not always be turned off for Send and ControlSend. Even when it is successfully turned off, it might not get turned back on after the keys are sent. This function is rarely used because the default behavior is best in most cases. This setting is ignored by blind mode and text mode; that is, the state of CapsLock is not changed in those cases. The built-in variable A_StoreCapsLockMode contains the current setting (1 or 0). Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related SetCaps/Num/ScrollLockState, Send, ControlSend Examples Causes the state of CapsLock not to be changed at all. SetStoreCapsLockMode False SetTimer - Syntax & Usage | AutoHotkey v2 SetTimer Causes a function to be called automatically and repeatedly at a specified time interval. SetTimer Function, Period, Priority Parameters Function Type: Function Object The function object to

call. A reference to the function object is kept in the script's list of timers, and is not released unless the timer is deleted. This occurs automatically for run-once timers, but can also be done by calling SetTimer with a Period of 0. If Function is omitted, SetTimer will operate on the timer which launched the current thread, if any. For example, SetTimer , 0 can be used inside a timer function to mark the timer for deletion, while SetTimer , 1000 would update the current timer's Period. Note: Passing an empty variable or an expression which results in an empty value is considered an error. This parameter must be either given a non-empty value or completely omitted. Period Type: Integer The absolute value of this parameter is used as the approximate number of milliseconds that must pass before the timer is executed. The timer will be automatically reset. It can be set to repeat automatically or run only once: If Period is greater than 0, the timer will automatically repeat until it is explicitly disabled by the script. If Period is less than 0, the timer will run only once. For example, specifying -100 would call Function 100 ms from now then delete the timer as though SetTimer Function, 0 had been used. If Period is 0, the timer is marked for deletion. If a thread started by this timer is still running, the timer is deleted after the thread finishes (unless it has been reenabled); otherwise, it is deleted immediately. In any case, the timer's previous Period and Priority are not retained. The absolute value of Period must be no larger than 4294967295 ms (49.7 days). Default: If this parameter is omitted and: 1) the timer does not exist: it will be created with a period of 250. 2) the timer already exists: it will be reset at its former period unless a Priority is specified. Priority Type: Integer This optional parameter is an integer between -2147483648 and 2147483647 (or an expression) to indicate this timer's thread priority. If omitted, 0 will be used. See Threads for details. To change the priority of an existing timer without affecting it in any other way, omit Period. Remarks Timers are useful because they run asynchronously, meaning that they will run at the specified frequency (interval) even when the script is waiting for a window, displaying a dialog, or busy with another task. Examples of their many uses include taking some action when the user becomes idle (as reflected by A_TimeIdle) or closing unwanted windows the moment they appear. Although timers may give the illusion that the script is performing more than one task simultaneously, this is not the case. Instead, timed functions are treated just like other threads: they can interrupt or be interrupted by another thread, such as a hotkey subroutine. See Threads for details. Whenever a timer is created or updated with a new period, its function will not be called right away; its time period must expire first. If you wish the timer's first execution to be immediate, call the timer's function directly (however, this will not start a new thread like the timer itself does; so settings such as SendMode will not start off at their defaults). Reset: If SetTimer is used on an existing timer, the timer is reset (unless Priority is specified and Period is omitted); in other words, the entirety of its period must elapse before its function will be called again. Timer precision: Due to the granularity of the OS's time-keeping system, Period is typically rounded up to the nearest multiple of 10 or 15.6 milliseconds (depending on the type of hardware and drivers installed). A shorter delay may be achieved via Loop+Sleep as demonstrated at DllCall+timeBeginPeriod+Sleep. Reliability: A timer might not be able to run at the expected time under the following conditions: Other applications are putting a heavy load on the CPU. The timer's function is still running when the timer period expires again. There are too many other competing timers. The timer has been interrupted by another thread, namely another timed function, hotkey subroutine, or custom menu item (this can be avoided via Critical). If this happens and the interrupting thread takes a long time to finish, the interrupted timer will be effectively disabled for the duration. However, any other timers will continue to run by interrupting the thread that interrupted the first timer. The script is uninterruptible as a result of Critical or Thread "Interrupt/Priority". During such times, timers will not run. Later, when the script becomes interruptible again, any overdue timer will run once as soon as possible and then resume its normal schedule. Although timers will operate when the script is suspended, they will not run if the current thread has Thread "NoTimers" in effect or whenever any thread is paused. In addition, they do not operate when the user is navigating through one of the script's menus (such as the tray icon menu or a menu bar). Because timers operate by temporarily interrupting the script's current activity, their functions should be kept short (so that they finish quickly) whenever a long interruption would be undesirable. Other remarks: A temporary timer might often be disabled by its own function (see examples at the bottom of this page). Whenever a function is called by a timer, it starts off fresh with the default values for settings such as SendMode. These defaults can be changed during script startup. If hotkey response time is crucial (such as in games) and the script contains any timers whose functions take longer than about 5 ms to execute, use the following function to avoid any chance of a 15 ms delay. Such a delay would otherwise happen if a hotkey is pressed at the exact moment a timer thread is in its period of uninterruptibility: Thread "interrupt", 0 ; Make all threads always-interruptible. If a timer is disabled while its function is currently running, that function will continue until it completes. The KeyHistory feature shows how many timers exist and how many are currently enabled. Related Threads, Thread (function), Critical, Function Objects Examples Closes unwanted windows whenever they appear. SetTimer CloseMailWarnings, 250 CloseMailWarnings() { WinClose "Microsoft Outlook", "A timeout occured while communicating" WinClose "Microsoft Outlook", "A connection to the server could not be established" } Waits for a certain window to appear and then alerts the user. SetTimer Alert1, 500 Alert1() { if not WinExist("Video Conversion", "Process Complete") return ; Otherwise: SetTimer , 0 ; i.e. the timer turns itself off here. MsgBox "The video conversion is finished." } Detects single, double, and triple-presses of a hotkey. This allows a hotkey to perform a different operation depending on how many times you press it. #c:: KeyWinC(ThisHotkey) ; This is a named function hotkey. { static winc_presses := 0 if winc_presses > 0 ; SetTimer already started, so we log the keypress instead. { winc_presses += 1 return } ; Otherwise, this is the first press of a new series. Set count to 1 and start ; the timer: winc_presses := 1 SetTimer After400, -400 ; Wait for more presses within a 400 millisecond window. After400() ; This is a nested function. { if winc_presses = 1 ; The key was pressed once. { Run "m:\" ; Open a folder. } else if winc_presses = 2 ; The key was pressed twice. { Run "m:\multimedia" ; Open a different folder. } else if winc_presses > 2 { MsgBox "Three or more clicks detected." } ; Regardless of which action above was triggered, reset the count to ; prepare for the next series of presses: winc_presses := 0 } } Uses a method as the timer function. counter := SecondCounter() counter.Start Sleep 5000 counter.Stop Sleep 2000 ; An example class for counting the seconds... class SecondCounter { __New() { this.interval := 1000 this.count := 0 ; Tick() has an implicit parameter "this" which is a reference to ; the object, so we need to create a function which encapsulates ; "this" and the method to call: this.timer := ObjBindMethod(this, "Tick") } Start() { SetTimer this.timer, this.interval ToolTip "Counter started" } Stop() { ; To turn off the timer, we must pass the same object as before: SetTimer this.timer, 0 ToolTip "Counter stopped at " this.count } ; In this example, the timer calls this method: Tick() { ToolTip ++this.count } } Tips relating to the above example: We can also use this.timer := this.Tick.Bind(this). When this.timer is called, it will effectively invoke tick_function.Call(this), where tick_function is the function object which implements that method. By contrast, ObjBindMethod produces an object which invokes this.Tick(). If we rename Tick to Call, we can just use this directly instead of this.timer. However, ObjBindMethod is useful when the object has multiple methods which should be called by different event sources, such as hotkeys, menu items, GUI controls, etc. If the timer is being modified or deleted from within a function/method called by the timer, it may be easier to omit the Function parameter. In some cases this avoids the need to retain the original object which was passed to SetTimer. SetTitleMatchMode - Syntax & Usage | AutoHotkey v2 SetTitleMatchMode Sets the matching behavior of the WinTitle parameter in built-in functions such as WinWait. SetTitleMatchMode MatchMode SetTitleMatchMode Speed Parameters MatchMode Type: Integer or String Specify one of the following digits or the word RegEx: 1 = A window's title must start with the specified WinTitle to be a match. 2 = A window's title can contain WinTitle anywhere inside it to be a match. 3 = A window's title must exactly match WinTitle to be a match. RegEx = Changes WinTitle, WinText, ExcludeTitle, and ExcludeText to accept regular expressions. For example: WinActivate "Untitled.*Notepad". RegEx also applies to ahk_class and ahk_exe; for example, ahk_class IEFrame searches for any window whose class name contains IEFrame anywhere (this is because by default, regular expressions find a match anywhere in the target string). For WinTitle, each component is separate. For example, in "i)^untitled ahk_class i)^notepad$ ahk_pid " mypid, "i)^untitled" and "i)^notepad$" are separate regex patterns and mypid is always compared numerically (it is not a regex pattern). For WinText, each text element (i.e. each control's text) is matched against the RegEx separately. Therefore, it is not possible to have a match span more than one text element. The modes above also affect ExcludeTitle in the same way as WinTitle. For example, mode 3 requires that a window's title exactly match ExcludeTitle for that window to be excluded. Speed Type: String One of the following words to specify how the WinText and ExcludeText parameters should be matched: Fast: This is the default behavior. Performance may be substantially better than Slow, but certain types of controls are not detected. For instance, text is typically detected within Static and Button controls, but not Edit controls, unless they are owned by the script. Slow: Can be much slower, but works with all controls which respond to the WM_GETTEXT message. Return Value Type: Integer or String This function returns the previous value of whichever setting was changed (either A_TitleMatchMode or A_TitleMatchModeSpeed). Remarks This function affects the behavior of all windowing functions, e.g. WinExist and WinActivate. WinGetText is affected in the same way as other functions, but it always uses the Slow method to retrieve text. If unspecified, TitleMatchMode defaults to 2 and fast. If a window group is used, the current title match mode applies to each individual rule in the group. Generally, the slow mode should be used only if the target window cannot be uniquely identified by its title and fast-mode text. This is because the slow mode can be extremely slow if there are any application windows that are busy or "not responding". Window Spy has an option for Slow TitleMatchMode so that its easy to determine whether the Slow mode is needed. If you wish to change both attributes, run the function twice as in this example: SetTitleMatchMode 2 SetTitleMatchMode "slow" The built-in variables A_TitleMatchMode and A_TitleMatchModeSpeed contain the current settings. Regardless of the current TitleMatchMode, WinTitle, WinText, ExcludeTitle and ExcludeText are case sensitive. The only exception is the case-insensitive option of the RegEx mode; for example: i)untitled - notepad. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related SetWinDelay, WinExist, WinActivate, RegExMatch Examples Forces windowing functions to operate upon windows whose titles contain WinTitle at the beginning instead of anywhere. SetTitleMatchMode 1 Allows windowing functions to detect more control types, but with lower performance. Note that Slow/Fast can be set independently of all the other modes. SetTitleMatchMode "Slow" SetWinDelay - Syntax & Usage | AutoHotkey v2 SetWinDelay Sets the delay that will occur after each windowing function, such as WinActivate. SetWinDelay Delay Parameters Delay Type: Integer Time in milliseconds. Use -1 for no delay at all and 0 for the smallest possible delay. If unset, the default delay is 100. Return Value Type: Integer This function returns the previous setting. Remarks A short delay (sleep) is done automatically after every windowing function except WinActive and WinExist. This is done to improve the reliability of scripts because a window sometimes needs a period of "rest" after

being created, activated, minimized, etc. so that it has a chance to update itself and respond to the next function that the script may attempt to send to it. Although a delay of -1 (no delay at all) is allowed, it is recommended that at least 0 be used, to increase confidence that the script will run correctly even when the CPU is under load. A delay of 0 internally executes a Sleep(0), which yields the remainder of the script's timeslice to any other process that may need it. If there is none, Sleep(0) will not sleep at all. If the CPU is slow or under load, or if window animation is enabled, higher delay values may be needed. The built-in variable A_WinDelay contains the current setting. Every newly launched thread (such as a hotkey, custom menu item, or timed subroutine) starts off fresh with the default setting for this function. That default may be changed by using this function during script startup. Related SetControlDelay, SetKeyDelay, SetMouseDelay, SendMode Examples Causes a delay of 10 ms to occur after each windowing function. SetWinDelay 10 SetWorkingDir - Syntax & Usage | AutoHotkey v2 SetWorkingDir Changes the script's current working directory. SetWorkingDir DirName Parameters DirName Type: String The name of the new working directory, which is assumed to be a subfolder of the current A_WorkingDir if an absolute path isn't specified. Error Handling An OSError is thrown on failure. Remarks The script's working directory is the default directory that is used to access files and folders when an absolute path has not been specified. In the following example, the file My Filename.txt is assumed to be in A_WorkingDir: FileAppend "A Line of Text", "My Filename.txt". The script's working directory defaults to A_ScriptDir, regardless of how the script was launched. By contrast, the value of A_InitialWorkingDir is determined by how the script 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. Once changed, the new working directory is instantly and globally in effect throughout the script. All interrupted, paused, and newly launched threads are affected, including Timers. Related A_WorkingDir, A_InitialWorkingDir, A_ScriptDir, DirSelect Examples Changes the script's current working directory. SetWorkingDir "D:\My Folder\Temp" Forces the script to use the folder it was initially launched from as its working directory. SetWorkingDir A_InitialWorkingDir Shutdown - Syntax & Usage | AutoHotkey v2 Shutdown Shuts down, restarts, or logs off the system. Shutdown Flag Parameters Flag Type: Integer A combination (sum) of the following numbers: 0 = Logoff 1 = Shutdown 2 = Reboot 4 = Force 8 = Power down Add the required values together. For example, to shutdown and power down the flag would be 9 (shutdown + power down = 1 + 8 = 9). The "Force" value (4) forces all open applications to close. It should only be used in an emergency because it may cause any open applications to lose data. The "Power down" value (8) shuts down the system and turns off the power. Remarks To have the system suspend or hibernate, see example #2 at the bottom of this page. To turn off the monitor, see SendMessage example #1. On a related note, a script can detect when the system is shutting down or the user is logging off via OnExit. Related Run, ExitApp, OnExit Examples Forces a reboot (reboot + force = 2 + 4 = 6). Shutdown 6 Calls the Windows API function "SetSuspendState" to have the system suspend or hibernate. Note that the second parameter may have no effect at all on newer systems. ; Parameter #1: Pass 1 instead of 0 to hibernate rather than suspend. ; Parameter #2: Pass 1 instead of 0 to suspend immediately rather than asking each application for permission. ; Parameter #3: Pass 1 instead of 0 to disable all wake events. DllCall("PowrProf\SetSuspendState", "Int", 0, "Int", 0, "Int", 0) Sleep - Syntax & Usage | AutoHotkey v2 Sleep Waits the specified amount of time before continuing. Sleep Delay Parameters Delay Type: Integer The amount of time to pause (in milliseconds) between 0 and 2147483647 (24 days). Remarks Due to the granularity of the OS's time-keeping system, Delay is typically rounded up to the nearest multiple of 10 or 15.6 milliseconds (depending on the type of hardware and drivers installed). To achieve a shorter delay, see Examples. The actual delay time might wind up being longer than what was requested if the CPU is under load. This is because the OS gives each needy process a slice of CPU time (typically 20 milliseconds) before giving another timeslice to the script. A delay of 0 yields the remainder of the script's current timeslice to any other processes that need it (as long as they are not significantly lower in priority than the script). Thus, a delay of 0 produces an actual delay between 0 and 20ms (or more), depending on the number of needy processes (if there are no needy processes, there will be no delay at all). However, a Delay of 0 should always wind up being shorter than any longer Delay would have been. While sleeping, new threads can be launched via hotkey, custom menu item, or timer. Sleep -1: A delay of -1 does not sleep but instead makes the script immediately check its message queue. This can be used to force any pending interruptions to occur at a specific place rather than somewhere more random. See Critical for more details. Related SetKeyDelay, SetMouseDelay, SetControlDelay, SetWinDelay Examples Waits 1 second before continuing execution. Sleep 1000 Waits 30 minutes before continuing execution. MyVar := 30 * 60000 ; 30 means minutes and times 60000 gives the time in milliseconds. Sleep MyVar ; Sleep for 30 minutes. Demonstrates how to sleep for less time than the normal 10 or 15.6 milliseconds. Note: While a script like this is running, the entire operating system and all applications are affected by timeBeginPeriod below. SleepDuration := 1 ; This can sometimes be finely adjusted (e.g. 2 is different than 3) depending on the value below. TimePeriod := 3 ; Try 7 or 3. See comment below. ; On a PC whose sleep duration normally rounds up to 15.6 ms, try TimePeriod=7 to allow ; somewhat shorter sleeps, and try TimePeriod=3 or less to allow the shortest possible sleeps. DllCall("Winmm\timeBeginPeriod", "UInt", TimePeriod) ; Affects all applications, not just this script's DllCall("Sleep"...), but does not affect SetTimer. Iterations := 50 StartTime := A_TickCount Loop Iterations DllCall("Sleep", "UInt", SleepDuration) ; Must use DllCall instead of the Sleep function. DllCall("Winmm\timeEndPeriod", "UInt", TimePeriod) ; Should be called to restore system to normal. MsgBox "Sleep duration = " . (A_TickCount - StartTime) / Iterations Sort - Syntax & Usage | AutoHotkey v2 Sort Arranges a variable's contents in alphabetical, numerical, or random order (optionally removing duplicates). SortedString := Sort(String , Options, Function) Parameters String Type: String The string to sort. Options Type: String See list below. Function Type: Function Object An optional function object for custom sorting. Return Value Type: String This function returns the sorted version of the specified string. Options A string of zero or more of the following letters (in any order, with optional spaces in between): C, C1 or COn: Case sensitive sort (ignored if the N option is also present). C0 or COff: Case insensitive sort. The uppercase letters A-Z are considered identical to their lowercase counterparts for the purpose of the sort. This is the default mode if none of the other case sensitivity options are used. CL or CLocale: Case insensitive sort based on the current user's locale. For example, most English and Western European locales treat the letters A-Z and ANSI letters like Ä and Ü as identical to their lowercase counterparts. This method also uses a "word sort", which treats hyphens and apostrophes in such a way that words like "coop" and "co-op" stay together. Depending on the content of the items being sorted, the performance will be 1 to 8 times worse than the default method of insensitivity. CLogical: Like CLocale, but digits in the strings are considered as numerical content rather than text. For example, "A2" is considered less than "A10". However, if two numbers differ only by the presence of a leading zero, the string with leading zero may be considered less than the other string. The exact behavior may differ between OS versions. Dx: Specifies x as the delimiter character, which determines where each item begins and ends. The delimiter is always case-sensitive. If this option is not present, x defaults to linefeed (`n). In most cases this will work even if lines end with CR+LF (`r`n), but the carriage return (`r) is included in comparisons and therefore affects the sort order. For example, B`r`nA will sort as expected, but A`r`nA`t`r`nB will place A`t`r before A`r. N: Numeric sort: Each item is assumed to be a number rather than a string (for example, if this option is not present, the string 233 is considered to be less than the string 40 due to alphabetical ordering). Both decimal and hexadecimal strings (e.g. 0xF1) are considered to be numeric. Strings that do not start with a number are considered to be zero for the purpose of the sort. Numbers are treated as 64-bit floating point values so that the decimal portion of each number (if any) is taken into account. Pn: Sorts items based on character position n (do not use hexadecimal for n). If this option is not present, n defaults to 1, which is the position of the first character. The sort compares each string to the others starting at its nth character. If n is greater than the length of any string, that string is considered to be blank for the purpose of the sort. When used with option N (numeric sort), the string's character position is used, which is not necessarily the same as the number's digit position. R: Sorts in reverse order (alphabetically or numerically depending on the other options). Random: Sorts in random order. This option causes all other options except D, Z, and U to be ignored (though N, C, and CL still affect how duplicates are detected). Examples: MyVar := Sort(MyVar, "Random") MyVar := Sort(MyVar, "Random Z D|") U: Removes duplicate items from the list so that every item is unique. If the C option is in effect, the case of items must match for them to be considered identical. If the N option is in effect, an item such as 2 would be considered a duplicate of 2.0. If either the Pn or \ (backslash) option is in effect, the entire item must be a duplicate, not just the substring that is used for sorting. If the Random option or custom sorting is in effect, duplicates are removed only if they appear adjacent to each other as a result of the sort. For example, when "A|B|A" is sorted randomly, the result could contain either one or two A's. Z: To understand this option, consider a variable that contains RED`nGREEN`nBLUE`n. If the Z option is not present, the last linefeed (`n) is considered to be part of the last item, and thus there are only 3 items. But by specifying Z, the last `n (if present) will be considered to delimit a blank item at the end of the list, and thus there are 4 items (the last being blank). \: Sorts items based on the substring that follows the last backslash in each. If an item has no backslash, the entire item is used as the substring. This option is useful for sorting bare filenames (i.e. excluding their paths), such as the example below, in which the AAA.txt line is sorted above the BBB.txt line because their directories are ignored for the purpose of the sort: C:\BBB\AAA.txt C:\AAA\BBB.txt Note: Options N and P are ignored when the backslash option is present. Custom Sorting Custom sorting is achieved by providing the Function parameter with a function or function object which compares any two items in the list. The function must accept two or three parameters: MyFunction(first, second , offset) When the function deems the first parameter to be greater than the second, it should return a positive integer; when it deems the two parameters to be equal, it should return 0, "", or nothing; otherwise, it should return a negative integer. If a decimal point is present in the returned value, that part is ignored (i.e. 0.8 is the same as 0). If present, the third parameter receives the offset (in characters) of the second item from the first as seen in the original/unsorted list (see examples). The function uses the same global (or thread-specific) settings as the Sort function that called it. Note: All options except D, Z, and U are ignored when a Function is present (though N, C, and CL still affect how duplicates are detected). Remarks This function is typically used to sort a variable that contains a list of lines, with each line ending in a linefeed character (`n). One way to get a list of lines into a variable is to load an entire file via FileRead. If a large variable was sorted and later its contents are no longer needed, you can free its memory by making it blank, e.g. MyVar := "". Related FileRead, file-reading loop, parsing loop, StrSplit, CallbackCreate, A_Clipboard Examples Sorts a comma-separated list of numbers. MyVar := "5,3,7,9,1,13,999,-4" MyVar := Sort(MyVar, "N D,") ; Sort numerically, use comma as delimiter. MsgBox MyVar ; The result is -4,1,3,5,7,9,13,999 Sorts the contents of a file. Contents := FileRead("C:\Address List.txt") FileDelete "C:\Address List (alphabetical).txt" FileAppend Sort(Contents), "C:\Address List (alphabetical).txt" Contents := "" ; Free the memory. Makes a hotkey to copy files from an open

Explorer window and put their sorted filenames onto the clipboard. #c:: ; Win+C { A_Clipboard := "" ; Must be blank for detection to work. Send "^c" if !ClipWait(2) return MsgBox "Ready to be pasted:`n" Sort(A_Clipboard) } Demonstrates custom sorting via a callback function. MyVar := "This`nis`nan`nexample`nstring`nto`nbe`nsorted" MsgBox Sort(MyVar,, LengthSort) LengthSort(a1, a2, *) { a1 := StrLen(a1), a2 := StrLen(a2) return a1 > a2 ? 1 : a1 < a2 ? -1 : 0 ; Sorts according to the lengths determined above. } MyVar := "5,3,7,9,1,13,999,-4" MsgBox Sort(MyVar, "D,", IntegerSort) IntegerSort(a1, a2, *) { return a1 - a2 ; Sorts in ascending numeric order. This method works only if the difference is never so large as to overflow a signed 64-bit integer. } MyVar := "1,2,3,4" MsgBox Sort(MyVar, "D,", ReverseDirection) ; Reverses the list so that it contains 4,3,2,1 ReverseDirection(a1, a2, offset) { return offset ; Offset is positive if a2 came after a1 in the original list; negative otherwise. } MyVar := "a bbb cc" ; Sorts in ascending length order; uses a fat arrow function: MsgBox Sort(MyVar, "D ", (a,b,*) => StrLen(a) - StrLen(b)) Sound Functions | AutoHotkey v2 Sound Functions Applies to: SoundGetVolume / SoundSetVolume SoundGetMute / SoundSetMute SoundGetName SoundGetInterface Other sound-related functions: SoundBeep SoundPlay Endpoint Devices The "devices" referenced by the SoundGet and SoundSet functions are audio endpoint devices. A single device driver or physical device often has multiple endpoints, such as for different types of output or input. For example: NameDescription Speakers (Example HD Audio)The main analog outputs of this device (uses multiple jacks in the case of surround sound). Digital Output (Example HD Audio)An optical or coaxial digital output. Microphone (Example HD Audio)Captures audio through a microphone jack. Stereo Mix (Example HD Audio)Captures whatever audio is being output to the Speakers endpoint. Device names typically consist of an endpoint name such as "Speakers" followed by the name of the audio driver in parentheses. Scripts may use the full name or just the leading part of the name, such as "Mic" or "Microphone". An audio driver has a fixed name, but endpoint names may be changed by an administrator at any time via the Sound control panel. Devices are listed in the Sound control panel, which can be opened by running mmsys.cpl from the command line, or via the Run dialog (Win+R) or the Run function. By default, the control panel only lists devices that are enabled and plugged in (if applicable), but this can be changed via the right-click menu. AutoHotkey detects devices which are not plugged in, but does not detect disabled devices. Components Components are as shown on the Levels tab of the sound device's properties dialog. In this example, the master controls are at the top, followed by the first four components: Microphone, FrontMic, Line In, and Side. All have volume and mute controls, except the fourth component, which only has volume. A sound device's properties dialog can be opened via the Sound control panel. Audio drivers are capable of exposing other controls, such as bass and treble. However, common audio drivers tend to have only volume and mute controls, or no components at all. Volume and mute controls are supported directly through SoundGetVolume, SoundSetVolume, SoundGetMute and SoundSetMute. All other controls are supported only indirectly, through SoundGetInterface and ComCall. Advanced Details Components are discovered through the DeviceTopology API, which exposes a graph of Connectors and Subunits. Each component shown above has a Connector, and it is the Connector that defines the component's name. Each control (such as volume or mute) is represented by a Subunit which sits between the Connector and the endpoint. Data "flows" from or to the Connector and is altered as it flows through each Subunit, such as to adjust volume or suppress (mute) all sound. The SoundGet and SoundSet functions identify components by walking the device topology graph and counting Connectors with the given name (or all Connectors if no name is given). Once the matching Connector is found, a control interface (such as IAudioVolumeLevel or IAudioMute) is retrieved by querying each Subunit on that specific branch of the graph, starting nearest the Connector. Subunits which apply to multiple Connectors are excluded - such as Subunits which correspond to the master volume and mute controls. A Connector is counted if (and only if) it has at least one Subunit of its own, even if the Subunit is not of the requested type. In practice, the end result is that the available components are as listed on the Levels tab, and in the same order. However, this process is based on observation, trial and error, so might not be 100% accurate. Common Parameters Component Type: String or Integer One of the following: The index of a component, where 1 is the first component. The full display name of a component (case-insensitive). As above, but followed by a colon and integer, where 1 is the first occurrence of a component with that name. For example, "Line In:2" uses the second component named "Line In". This is necessary only when Component would otherwise be ambiguous, such as when multiple components exist with the same name, or the display name is empty, an integer or contains a colon. If omitted or blank, the function targets the master volume/mute controls or an interface that can be returned by IMMDevice::Activate. If only an index is specified, the display names are ignored. For example, 1, "1" and ":1" use the first component regardless of name, whereas "" uses the master controls. If the sound device lacks the specified Component, a TargetError is thrown. Device Type: String or Integer One of the following: A number (integer) between 1 and the total number of supported devices. The display name of a device; either the full name or a leading part (case-insensitive). For example, "Speakers" or "Speakers (Example HD Audio)". As above, but followed by a colon and integer, where 1 is the first device with a matching name. For example, "Speakers:2" indicates the second device which has a name starting with "Speakers". This is necessary only when Device would otherwise be ambiguous, such as when multiple devices exist with the same name, or the display name contains a colon. If omitted or blank, it defaults to the system's default device for playback (which is not necessarily device 1). The soundcard analysis script may help determine which name and/or number to use. Examples Soundcard Analysis. Use the following script to discover the available audio device and component names and whether each device or component supports volume and/or mute controls. It displays the results in a simple ListView. The current volume and mute settings are shown if they can be retrieved, but are not updated in realtime. scGui := Gui(, "Sound Components") scLV := scGui.Add('ListView', "w600 h400" , ["Component", "#", "Device", "Volume", "Mute"]) devMap := Map() loop { ; For each loop iteration, try to get the corresponding device. try devName := SoundGetName(, dev := A_Index) catch ; No more devices. break ; Qualify names with ":index" where needed. devName := Qualify(devName, devMap, dev) ; Retrieve master volume and mute setting, if possible. vol := mute := "" try vol := Round(SoundGetVolume(, dev), 2) try mute := SoundGetMute(, dev) ; Display the master settings only if at least one was retrieved. if vol != "" || mute != "" scLV.Add("", "", dev, devName, vol, mute) ; For each component, first query its name. cmpMap := Map() loop { try cmpName := SoundGetName(cmp := A_Index, dev) catch break ; Retrieve this component's volume and mute setting, if possible. vol := mute := "" try vol := Round(SoundGetVolume(cmp, dev), 2) try mute := SoundGetMute(cmp, dev) ; Display this component even if it does not support volume or mute, ; since it likely supports other controls via SoundGetInterface(). scLV.Add("", Qualify(cmpName, cmpMap, A_Index), dev, devName, vol, mute) } } loop 5 scLV.ModifyCol(A_Index, 'AutoHdr Logical') scGui.Show() ; Qualifies full names with ":index" when needed. Qualify(name, names, overallIndex) { if name = '' return overallIndex key := StrLower(name) index := names.Has(key) ? ++names[key] : (names[key] := 1) return (index > 1 || InStr(name, ':') || IsInteger(name)) ? name ':' index : name } SoundBeep - Syntax & Usage | AutoHotkey v2 SoundBeep Emits a tone from the PC speaker. SoundBeep Frequency, Duration Parameters Frequency Type: Integer The frequency of the sound. It should be a number between 37 and 32767. If omitted, the frequency will be 523. Duration Type: Integer The duration of the sound, in milliseconds. If omitted, the duration will be 150. Remarks The script waits for the sound to finish before continuing. In addition, system responsiveness might be reduced during sound production. If the computer lacks a sound card, a standard beep is played through the PC speaker. To produce the standard system sounds instead of beeping the PC Speaker, see the asterisk mode of SoundPlay. Related SoundPlay Examples Plays the default pitch and duration. SoundBeep Plays a higher pitch for half a second. SoundBeep 750, 500 SoundGetInterface - Syntax & Usage | AutoHotkey v2 SoundGetInterface Retrieves a native COM interface of a sound device or component. InterfacePtr := SoundGetInterface(IID, Component, Device) Parameters IID Type: String An interface identifier (GUID) in the form "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, an interface implemented by the device itself is retrieved. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Return Value Type: Integer On success, the return value is an interface pointer. If the interface is not supported, the return value is zero. Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found OSErrorSystem-generated. Remarks The interface is retrieved from one of the following sources: If Component is omitted, IMMDevice::Activate is called to retrieve the interface. QueryInterface is called for the Connector identified by Component, and if successful, the interface pointer is returned. This can be used to retrieve the IPart or IConnector interface of the Connector. IPart::Activate is called for each Subunit unique to the given Component. For example, IID can be "{7FB7B48F-531D-44A2-BCB3-5AD5A134B3DC}" to retrieve the IAudioVolumeLevel interface, which provides access to per-channel volume level controls. Once the interface pointer is retrieved, ComCall can be used to call its methods. Refer to the Windows SDK header files to identify the correct method index. The interface pointer must be released by passing it to ObjRelease when it is no longer needed. This can be done by "wrapping" it with ComValue. The wrapped value (an object) can be passed directly to ComCall. Interface := ComValue(13, InterfacePtr) Related Sound Functions, DeviceTopology API Examples Peak meter. A tooltip is displayed with the current peak value, except when the peak value is zero (no sounds are playing). ; IAudioMeterInformation audioMeter := SoundGetInterface("{C02216F6-8C67-4B5B-9D00-D008E73E0064}") if audioMeter { try loop ; Until the script exits or an error occurs. { ; audioMeter->GetPeakValue(&peak) ComCall 3, audioMeter, "float*", &peak:=0 ToolTip peak > 0 ? peak : "" Sleep 15 } ObjRelease audioMeter } else MsgBox "Unable to get audio meter" SoundGetMute - Syntax & Usage | AutoHotkey v2 SoundGetMute Retrieves a mute setting of a sound device. Setting := SoundGetMute(Component, Device) Parameters Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, the master mute setting is retrieved. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Return Value Type: Integer (boolean) This function returns 0 (not muted) or 1 (muted). Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found Component doesn't support this control type OSErrorSystem-generated. Remarks To discover the capabilities of the sound devices installed on the system -- such as the names and available components -- run the soundcard analysis script. Related Sound Functions Examples Checks whether the default playback device is muted. master_mute := SoundGetMute() if master_mute MsgBox "The default playback

device is muted." else MsgBox "The default playback device is not muted." Checks whether "Line In pass-through" is muted. if SoundGetMute("Line In") = 0 MsgBox "Line In pass-through is not muted." Checks whether the microphone (recording) is muted. if SoundGetMute(, "Microphone") = 0 MsgBox "The microphone (recording) is not muted." SoundGetName - Syntax & Usage | AutoHotkey v2 SoundGetName Retrieves the name of a sound device or component. Name := SoundGetName(Component, Device) Parameters Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, the name of the device itself is retrieved. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Return Value Type: String The name of the device or component, which can be empty. Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found OSErrorSystem-generated. Related Sound Functions Examples Retrieves and reports the name of the default playback device. default_device := SoundGetName() MsgBox 'The default playback device is "' default_device '".' Retrieves and reports the name of the first device. device1 := SoundGetName(, 1) MsgBox 'Device 1 is "' device1 '".' Retrieves and reports the name of the first component. component1 := SoundGetName(1) MsgBox 'Component 1 is "' component1 '".' For a more complex example, see the soundcard analysis script. SoundGetVolume - Syntax & Usage | AutoHotkey v2 SoundGetVolume Retrieves a volume setting of a sound device. Setting := SoundGetVolume(Component, Device) Parameters Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, the master volume setting is retrieved. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Return Value Type: Float This function returns a floating point number between 0.0 and 100.0. Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found Component doesn't support this control type OSErrorSystem-generated. Remarks To discover the capabilities of the sound devices installed on the system -- such as the names and available components -- run the soundcard analysis script. Related Sound Functions Examples Retrieves and reports the master volume. master_volume := SoundGetVolume() MsgBox "Master volume is " master_volume " percent." Retrieves and reports the microphone listening volume. mic_volume := SoundGetVolume("Microphone") MsgBox "Microphone listening volume is " mic_volume " percent." Retrieves and reports the microphone recording volume. mic_volume := SoundGetVolume(, "Microphone") MsgBox "Microphone recording volume is " mic_volume " percent." SoundPlay - Syntax & Usage | AutoHotkey v2 SoundPlay Plays a sound, video, or other supported file type. SoundPlay Filename , Wait Parameters Filename Type: String The name of the file to be played, which is assumed to be in A_WorkingDir if an absolute path isn't specified. To produce standard system sounds, specify an asterisk followed by a number as shown below (note that the Wait parameter has no effect in this mode): *-1 = Simple beep. If the sound card is not available, the sound is generated using the speaker. *16 = Hand (stop/error) *32 = Question *48 = Exclamation *64 = Asterisk (info) Known limitation: Due to a quirk in Windows, WAV files with a path longer than 127 characters will not be played. To work around this, use other file types such as MP3 (with a path length of up to 255 characters) or use 8.3 short paths (see A_LoopFileShortPath how to retrieve such paths). Wait Type: String If omitted, the script's current thread will move on to the next statement(s) while the file is playing. To avoid this, specify 1 or the word WAIT, which causes the current thread to wait until the file is finished playing before continuing. Even while waiting, new threads can be launched via hotkey, custom menu item, or timer. Known limitation: If the WAIT parameter is omitted, the OS might consider the playing file to be "in use" until the script closes or until another file is played (even a nonexistent file). Error Handling An exception is thrown on failure. Remarks All Windows OSes should be able to play .wav files. However, other files (.mp3, .avi, etc.) might not be playable if the right codecs or features aren't installed on the OS. If a file is playing and the current script plays a second file, the first file will be stopped so that the second one can play. On some systems, certain file types might stop playing even when an entirely separate script plays a new file. To stop a file that is currently playing, use SoundPlay on a nonexistent filename as in this example: try SoundPlay "Nonexistent.avi". If the script is exited, any currently-playing file that it started will stop. Related SoundBeep, Sound Functions, MsgBox, Threads Examples Plays a sound file located in the Windows directory. SoundPlay A_WinDir "\Media\ding.wav" Generates a simple beep. If the sound card is not available, the sound is generated using the speaker. SoundPlay "*-1" SoundSetMute - Syntax & Usage | AutoHotkey v2 SoundSetMute Changes a mute setting of a sound device. SoundSetMute NewSetting , Component, Device Parameters NewSetting Type: Number or String Any positive number will turn on the setting (mute) and a zero will turn it off (unmute). However, if the number begins with a plus or minus sign, the setting will be toggled (set to the opposite of its current state). Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, the master mute setting is changed. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found Component doesn't support this control type OSErrorSystem-generated. Remarks An alternative way to toggle the master mute setting of the default playback device is to have the script send a keystroke, such as in the example below: Send "{Volume_Mute}" ; Mute/unmute the master volume. To discover the capabilities of the sound devices installed on the system -- such as the names and available components -- run the soundcard analysis script. Use SoundGetMute to retrieve the current mute setting. Related Sound Functions Examples Sets master mute (on). SoundSetMute true Toggles the master mute (sets it to the opposite state). SoundSetMute -1 Mutes Line In. SoundSetMute 1, "Line In" Mutes microphone recording. SoundSetMute 1, , "Microphone" SoundSetVolume - Syntax & Usage | AutoHotkey v2 SoundSetVolume Changes a volume setting of a sound device. SoundSetVolume NewSetting , Component, Device Parameters NewSetting Type: Number or String Percentage number between -100 and 100 inclusive (it can be a floating point number). If the number begins with a plus or minus sign, the current setting will be adjusted up or down by the indicated amount. Otherwise, the setting will be set explicitly to the level indicated by NewSetting. Component Type: String or Integer The component's display name and/or index. For example, 1, "Line in" or "Line in:2". If omitted or blank, the master volume setting is changed. For further details, see Component (Sound Functions). Device Type: String or Integer The device's display name and/or index. For example, 1, "Speakers", "Speakers:2" or "Speakers (Example HD Audio)". If this parameter is omitted, it defaults to the system's default device for playback (which is not necessarily device 1). For further details, see Device (Sound Functions). Error Handling One of the following exceptions may be thrown on failure: Error ClassMessage TargetErrorDevice not found Component not found Component doesn't support this control type OSErrorSystem-generated. Remarks An alternative way to adjust the volume is to have the script send volume-control keystrokes to change the master volume for the entire system, such as in the example below: Send "{Volume_Up}" ; Raise the master volume by 1 interval (typically 5%). Send "{Volume_Down 3}" ; Lower the master volume by 3 intervals. To discover the capabilities of the sound devices installed on the system -- such as the names and available components -- run the soundcard analysis script. SoundSetVolume attempts to preserve the existing balance when changing the volume level. Use SoundGetVolume to retrieve the current volume setting. Related Sound Functions Examples Sets the master volume to 50%. SoundSetVolume 50 Increases the master volume by 10%. SoundSetVolume "+10" Decreases the master volume by 10%. SoundSetVolume -10 Increases microphone recording volume by 20%. SoundSetVolume "+20", , "Microphone" SplitPath - Syntax & Usage | AutoHotkey v2 SplitPath Separates a file name or URL into its name, directory, extension, and drive. SplitPath Path , &OutFileName, &OutDir, &OutExtension, &OutNameNoExt, &OutDrive Parameters Path Type: String The file name or URL to be analyzed. Note that this function expects filename paths to contain backslashes (\) only and URLs to contain forward slashes (/) only. &OutFileName Type: VarRef A reference to the output variable in which to store the file name without its path. The file's extension is included. &OutDir Type: VarRef A reference to the output variable in which to store the directory of the file, including drive letter or share name (if present). The final backslash is not included even if the file is located in a drive's root directory. &OutExtension Type: VarRef A reference to the output variable in which to store the file's extension (e.g. TXT, DOC, or EXE). The dot is not included. &OutNameNoExt Type: VarRef A reference to the output variable in which to store the file name without its path, dot and extension. &OutDrive Type: VarRef A reference to the output variable in which to store the drive letter or server name of the file. If the file is on a local or mapped drive, the variable will be set to the drive letter followed by a colon (no backslash). If the file is on a network path (UNC), the variable will be set to the share name, e.g. \\Workstation01 Remarks Any of the output variables may be omitted if the corresponding information is not needed. If Path contains a filename that lacks a drive letter (that is, it has no path or merely a relative path), OutDrive will be made blank but all the other output variables will be set correctly. Similarly, if there is no path present, OutDir will be made blank; and if there is a path but no file name present, OutFileName and OutNameNoExt will be made blank. Actual files and directories in the file system are not checked by this function. It simply analyzes the provided string. Wildcards (* and ?) and other characters illegal in filenames are treated the same as legal characters, with the exception of colon, backslash, and period (dot), which are processed according to their nature in delimiting the drive letter, directory, and extension of the file. Support for URLs: If Path contains a colon-double-slash, such as https://domain.com or ftp://domain.com, OutDir is set to the protocol prefix + domain name + directory (e.g. https://domain.com/images) and OutDrive is set to the protocol prefix + domain name (e.g. https://domain.com). All other variables are set according to their definitions above. Related A_LoopFileExt, StrSplit, InStr, SubStr, FileSelect, DirSelect Examples Demonstrates different usages. FullFileName := "C:\My Documents\Address List.txt" ; To fetch only the bare filename from the above: SplitPath FullFileName, &name ; To fetch only its directory: SplitPath FullFileName,, &dir ; To fetch all info: SplitPath FullFileName, &name, &dir, &ext, &name_no_ext, &drive ; The above will set the variables as follows: ; name = Address List.txt ; dir = C:\My Documents ; ext = txt ; name_no_ext = Address List ; drive = C: StatusbarGetText - Syntax & Usage | AutoHotkey v2 StatusBarGetText Retrieves the text from a standard status bar control. Text := StatusBarGetText(Part#, WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters Part# Type: Integer Which part number of the bar to retrieve. Default 1, which is usually the part that contains the text of interest.

WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the text from a single part of the status bar control. Error Handling An exception is thrown in any of the following cases: TargetError: The target window could not be found or does not contain a standard status bar. OSError: There was a problem sending the SB_GETPARTS message, or no reply was received within 2000ms. OSError: Memory could not be allocated within the process which owns the status bar. Remarks This function attempts to read the first standard status bar on a window (Microsoft common control: msctls_statusbar32). Some programs use their own status bars or special versions of the MS common control, in which case the text cannot be retrieved. Rather than using this function in a loop, it is usually more efficient to use StatusBarWait, which contains optimizations that avoid the overhead of repeated calls to StatusBarGetText. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related StatusBarWait, WinGetTitle, WinGetText, ControlGetText Examples Retrieves and analyzes the text from the first part of a status bar. RetrievedText := StatusBarGetText(1, "Search Results") if InStr(RetrievedText, "found") MsgBox "Search results have been found." StatusBarWait - Syntax & Usage | AutoHotkey v2 StatusBarWait Waits until a window's status bar contains the specified string. StatusBarWait BarText, Timeout, Part#, WinTitle, WinText, Interval, ExcludeTitle, ExcludeText Parameters BarText Type: String The text or partial text for the which the function will wait to appear. Default is blank (empty), which means to wait for the status bar to become blank. The text is case sensitive and the matching behavior is determined by SetTitleMatchMode, similar to WinTitle below. To instead wait for the bar's text to change, either use StatusBarGetText in a loop, or use the RegEx example at the bottom of this page. Timeout Type: Integer or Float The number of seconds (can contain a decimal point) to wait before timing out, in which case the return value is 0 (false). Default is blank, which means the function will wait indefinitely. Part# Type: Integer Which part number of the bar to retrieve. Default 1, which is usually the part that contains the text of interest. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. Interval Type: Integer How often the status bar should be checked while the function is waiting (in milliseconds). Default is 50. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer (boolean) This function returns 1 (true) if a match was found or 0 (false) if the function timed out. Error Handling An exception is thrown in any of the following cases: TargetError: The target window could not be found or does not contain a standard status bar. OSError: There was a problem sending the SB_GETPARTS message, or no reply was received within 2000ms. OSError: Memory could not be allocated within the process which owns the status bar. Remarks StatusBarWait attempts to read the first standard status bar on a window (class msctls_statusbar32). Some programs use their own status bars or special versions of the MS common control. Such bars are not supported. Rather than using StatusBarGetText in a loop, it is usually more efficient to use StatusBarWait because it contains optimizations that avoid the overhead that repeated calls to StatusBarGetText would incur. StatusBarWait determines its target window before it begins waiting for a match. If that target window is closed, the function will stop waiting even if there is another window matching the specified WinTitle and WinText. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related StatusBarGetText, WinGetTitle, WinGetText, ControlGetText Examples Enters a new search pattern into an existing Explorer/Search window. if WinExist("Search Results") ; Sets the Last Found window to simplify the below. { WinActivate Send "{tab 2}!o*.txt{enter}" ; In the Search window, enter the pattern to search for. Sleep 400 ; Give the status bar time to change to "Searching". if StatusBarWait ("found", 30) MsgBox "The search successfully completed." else MsgBox "The function timed out." } Waits for the status bar of the active window to change. SetTitleMatchMode "RegEx" ; Accept regular expressions for use below. if WinExist("A") ; Set the last-found window to be the active window (for use below). { OrigText := StatusBarGetText() StatusBarWait "^(?!^\Q" OrigText "\E$)" ; This regular expression waits for any change to the text. } StrCompare - Syntax & Usage | AutoHotkey v2 StrCompare Compares two strings alphabetically. Result := StrCompare(String1, String2 , CaseSense) Parameters String1, String2 Type: String The strings to be compared. CaseSense Type: Integer or String One of the following values (defaults to 0 if omitted): "On" or 1 (True): The comparison is case sensitive. "Off" or 0 (False): The letters A-Z are considered identical to their lowercase counterparts. "Locale": The comparison is case insensitive according to the rules of the current user's locale. For example, most English and Western European locales treat not only the letters A-Z as identical to their lowercase counterparts, but also non-ASCII letters like Ä and Ü as identical to theirs. Locale is 1 to 8 times slower than Off depending on the nature of the strings being compared. "Logical": Like Locale, but digits in the strings are considered as numerical content rather than text. For example, "A2" is considered less than "A10". However, if two numbers differ only by the presence of a leading zero, the string with leading zero may be considered less than the other string. The exact behavior may differ between OS versions. Return Value Type: Integer To indicate the relationship between String1 and String2, this function returns one of the following: 0, if String1 is identical to String2 a positive integer, if String1 is greater than String2 a negative integer, if String1 is less than String2 To check for a specific relationship between the two strings, compare the result to 0. For example: a_less_than_b := StrCompare(a, b) < 0 a_greater_than_or_equal_to_b := StrCompare(a, b) >= 0 Remarks This function is commonly used for sort callbacks. Related Sort, VerCompare Examples Demonstrates the difference between a case insensitive and case sensitive comparison. MsgBox StrCompare("Abc", "abc") ; Returns 0 MsgBox StrCompare("Abc", "abc", true) ; Returns -1 StrGet - Syntax & Usage | AutoHotkey v2 StrGet Copies a string from a memory address or buffer, optionally converting it from a given code page. String := StrGet(Source , Length , Encoding) Parameters Source Type: Object or Integer A Buffer-like object containing the string, or the memory address of the string. Any object which implements Ptr and Size properties may be used, but this function is optimized for the native Buffer object. Passing an object with these properties ensures that the function does not read memory from an invalid location; doing so could cause crashes or other unpredictable behaviour. The string is not required to be null-terminated if a Buffer-like object is provided, or if the Length parameter is specified. Length Type: Integer The maximum number of characters to read. This can be omitted if the string is null-terminated. By default, StrGet only copies up to the first binary zero. 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. Note: Omitting Length when the string is not null-terminated may cause an access violation which terminates the program, or some other undesired result. Specifying an incorrect length may produce unexpected behaviour. Encoding Type: String or Integer The source encoding; for example, "UTF-8", "UTF-16" or "CP936". For numeric identifiers, the prefix "CP" can be omitted only if Length is specified. Specify an empty string or "CP0" to use the system default ANSI code page. Return Value Type: String This function returns the copied or converted string. If the source encoding was specified correctly, the return value always uses the native encoding. It is always null-terminated, but the null-terminator is not included in the return value's length except when Length is negative as described above. Error Handling A ValueError is thrown if invalid parameters are detected. An OSError is thrown if the conversion could not be performed. Remarks Note that the return value is always in the native encoding of the current executable, whereas Encoding specifies how to interpret the string read from the given Source. If no Encoding is specified, the string is simply copied without any conversion taking place. In other words, StrGet is used to retrieve text from a memory address or buffer, or convert it to a format the script can understand. If conversion between code pages is necessary, the length of the return value may differ from the length of the source string. Related String Encoding, StrPut, Binary Compatibility, FileEncoding, DllCall, Buffer object, VarSetStrCapacity Examples Either Length or Encoding may be specified directly after Source, but in those cases Encoding must be non-numeric. str := StrGet(address, "cp0") ; Code page 0, unspecified length str := StrGet(address, n, 0) ; Maximum n chars, code page 0 str := StrGet(address, 0) ; Maximum 0 chars (always blank) String - Syntax & Usage | AutoHotkey v2 String Converts a value to a string. StrValue := String(Value) Return Value Type: String This function returns the result of converting Value to a string, or Value itself if it is a string. If Value is a number, the default decimal formatting is used. If Value is an object, the return value is the result of calling Value.ToString(). If the object has no such method, a MethodError is thrown. This method is not implemented by default, so must be defined by the script. Related Type, Integer, Float, Values, Expressions, Is functions StrLen - Syntax & Usage | AutoHotkey v2 StrLen Retrieves the count of how many characters are in a string. Length := StrLen(String) Parameters String Type: String The string whose contents will be measured. Return Value Type: Integer This function returns the length of the specified string. Related InStr, SubStr, Trim, StrLower, StrUpper, StrPut, StrGet, StrReplace, StrSplit Examples Retrieves and reports the count of how many characters are in a string. StrValue := "The quick brown fox jumps over the lazy dog" MsgBox "The length of the string is " StrLen(StrValue) ; Result: 43 StrLower / StrUpper / StrTitle - Syntax & Usage | AutoHotkey v2 StrLower / StrUpper / StrTitle Converts a string to lowercase, uppercase or title case. NewString := StrLower(String) NewString := StrUpper (String) NewString := StrTitle(String) Parameters String Type: String The string to convert. Return Value Type: String These functions return the newly converted version of the specified string. Remarks To detect whether a character or string is entirely uppercase or lowercase, use the IsUpper, IsLower or RegExMatch function. For example: var := "abc" if isUpper(var) MsgBox "var is empty or contains only uppercase characters." if isLower(var) MsgBox "var is empty or contains only lowercase characters." if RegExMatch(var, "^[a-z]+$") MsgBox "var is not empty and contains only lowercase ASCII characters." if !RegExMatch (var, "[A-Z]") MsgBox "var does not contain any uppercase ASCII characters." Format can also be used for case conversions, as shown below: MsgBox Format ("{:U}, {:L} and {:T}", "upper", "LOWER", "title") Related InStr, SubStr, StrLen, StrReplace Examples Converts to lower case and stores the string "this is a test." in String1. String1 := "This is a test." String1 := StrLower(String1) ; i.e. output can be the same as input. Converts to upper case and stores the string "THIS IS A TEST." in String2. String2 := "This is a test." String2 := StrUpper(String2) Converts to title case and stores the string "This Is A Test." in String3. String3 := "This is a test." String3 := StrTitle(String3) StrPtr - Syntax & Usage | AutoHotkey v2 StrPtr Returns the current memory address of a string. Address := StrPtr (Value) Parameters Value Type: String Return Value Type: Integer This function returns the current memory address of Value. Remarks The lifetime of an

address and which operations are valid to perform on it depend on how Value was passed to this function. There are three distinct cases, shown as example code below. In all cases, if the string will not be modified, the return value can be passed directly to a DllCall function or SendMessage. Ptr := StrPtr(MyVar) If Value is a variable reference such as MyVar (and not a built-in variable), the return value is the memory address of the variable's internal string buffer. VarSetStrCapacity (&MyVar) returns the size of the buffer in characters, excluding the terminating null character. The address should be considered valid only until the variable is freed or has been reassigned by means of any of the assignment operators or by passing it to a built-in function. The address of the contents of a function's local variable is not valid after the function returns, as local variables are freed automatically. The address can be stored in a structure or another variable, and passed indirectly to DllCall or SendMessage or used in other ways, for as long as it remains valid as described above. The script may change the value of the string indirectly by passing the address to NumPut, DllCall or SendMessage. If the length of the string is changed this way, the variable's internal length property must be updated by calling VarSetStrCapacity(MyVar, -1). Ptr := StrPtr("literal string") The address of a literal string is valid until the program exits. The script should not attempt to modify the string. The address can be stored in a structure or another variable, and passed indirectly to DllCall or SendMessage or used in other ways. SendMessage 0x000C, 0, StrPtr(A_ScriptName " changed this title"),, "A" The address of a temporary string is valid only until evaluation of the overall expression or function call statement is completed, after which time it must not be used. For the example above, the address is valid until SendMessage returns. All of the following yield temporary strings: Concatenation. Built-in variables such as A_ScriptName. Functions which return strings. Accessing properties of an object, array elements or map elements. If not explicitly covered above, it is safe to assume the string is temporary. Related VarSetStrCapacity, DllCall, SendMessage, Buffer object, NumPut, NumGet StrPut - Syntax & Usage | AutoHotkey v2 StrPut Copies a string to a memory address or buffer, optionally converting it to a given code page. StrPut String , Encoding := None StrPut String, Target , Length , Encoding := None Parameters String Type: String Any string. If a number is given, it is automatically converted to a string. String is assumed to be in the native encoding. Target Type: Object or Integer A Buffer-like object or memory address to which the string will be written. Any object which implements Ptr and Size properties may be used, but this function is optimized for the native Buffer object. Passing an object with these properties ensures that the function does not write memory to an invalid location; doing so could cause crashes or other unpredictable behaviour. Note: If conversion between code pages is necessary, the required buffer size may differ from the size of the source string. For such cases, call StrPut with two parameters to calculate the required size. Length Type: Integer The maximum number of characters to write, including the null-terminator if required. If Length is zero or less than the projected length after conversion (or the length of the source string if conversion is not required), an exception is thrown. Length must not be omitted when Target is a plain memory address, unless the buffer size is known to be sufficient, such as if the buffer was allocated based on a previous call to StrPut with the same Source and Encoding. If Target is an object, specifying a Length that exceeds the buffer size calculated from Target.Size is considered an error, even if the converted string would fit within the buffer. Note: When Encoding is specified, Length should be the size of the buffer (in characters), not the length of String or a substring, as conversion may increase its length. Note: Length is measured in characters, whereas buffer sizes are usually measured in bytes, as is StrPut's return value. To specify the buffer size in bytes, use a Buffer-like object in the Target parameter. Encoding Type: String or Integer The target encoding; for example, "UTF-8", "UTF-16" or "CP936". For numeric identifiers, the prefix "CP" can be omitted only if Length is specified. Specify an empty string or "CP0" to use the system default ANSI code page. Return Value Type: Integer If Target is omitted, this function returns the required buffer size in bytes, including space for the null-terminator. If Target is specified, this function returns the number of bytes written. A null-terminator is written and included in the return value only when there is sufficient space; that is, it is omitted when Length or Target.Size (multiplied by the size of a character) exactly equals the length of the converted string. Error Handling A ValueError is thrown if invalid parameters are detected, such as if the converted string would be longer than allowed by Length or Target.Size. An OSError is thrown if the conversion could not be performed. Remarks Note that the String parameter is always assumed to use the native encoding of the current executable, whereas Encoding specifies the encoding of the string written to the given Target. If no Encoding is specified, the string is simply measured or copied without any conversion taking place. Related String Encoding, StrGet, Binary Compatibility, FileEncoding, DllCall, Buffer object, VarSetStrCapacity Examples Either Length or Encoding may be specified directly after Target, but in those cases Encoding must be non-numeric. StrPut(str, address, "cp0") ; Code page 0, unspecified buffer size StrPut(str, address, n, 0) ; Maximum n chars, code page 0 StrPut(str, address, 0) ; Unsupported (maximum 0 chars) StrPut may be called once to calculate the required buffer size for a string in a particular encoding, then again to encode and write the string into the buffer. The process can be simplified by utilizing this function. ; Returns a Buffer object containing the string. StrBuf(str, encoding) { ; Calculate required size and allocate a buffer. buf := Buffer(StrPut(str, encoding)) ; Copy or convert the string. StrPut(str, buf, encoding) return buf } StrReplace - Syntax & Usage | AutoHotkey v2 StrReplace Replaces the specified substring with a new string. ReplacedStr := StrReplace(Haystack, Needle , ReplaceText, CaseSense, &OutputVarCount, Limit) Parameters Haystack Type: String The string whose content is searched and replaced. Needle Type: String The string to search for. ReplaceText Type: String Needle will be replaced with this text. If omitted or blank, Needle will be replaced with blank (empty). In other words, it will be omitted from the return value. CaseSense Type: Integer or String One of the following values (defaults to 0 if omitted): "On" or 1 (True): The search is case sensitive. "Off" or 0 (False): The letters A-Z are considered identical to their lowercase counterparts. "Locale": The search is case insensitive according to the rules of the current user's locale. For example, most English and Western European locales treat not only the letters A-Z as identical to their lowercase counterparts, but also non-ASCII letters like Ä and Ü as identical to theirs. Locale is 1 to 8 times slower than Off depending on the nature of the strings being compared. &OutputVarCount Type: VarRef A reference to a output variable in which to store the number of replacements that occurred (0 if none). Limit Type: Integer If Limit is omitted, it defaults to -1, which replaces all occurrences of the pattern found in Haystack. Otherwise, specify the maximum number of replacements to allow. The part of Haystack to the right of the last replacement is left unchanged. Return Value Type: String This function returns the replaced version of the specified string. Remarks The built-in variables A_Space and A_Tab contain a single space and a single tab character, respectively. They are useful when searching for spaces and tabs alone or at the beginning or end of Needle. Related RegExReplace, InStr, SubStr, StrLen, StrLower, StrUpper Examples Removes all CR+LF's from the clipboard contents. A_Clipboard := StrReplace(A_Clipboard, "`r`n") Replaces all spaces with pluses. NewStr := StrReplace(OldStr, A_Space, "+") Removes all blank lines from the text in a variable. Loop { MyString := StrReplace(MyString, "`r`n`r`n", "`r`n",, &Count) if (Count = 0) ; No more replacements needed. break } StrSplit - Syntax & Usage | AutoHotkey v2 StrSplit Separates a string into an array of substrings using the specified delimiters. Array := StrSplit(String , Delimiters, OmitChars, MaxParts) Parameters String Type: String A string to split. Delimiters Type: String or Array of Strings If this parameter is blank or omitted, each character of the input string will be treated as a separate substring. Otherwise, Delimiters can be either a single string or an array of strings, each of which is used to determine where the boundaries between substrings occur. Since the delimiters are not considered to be part of the substrings themselves, they are never included in the returned array. Also, if there is nothing between a pair of delimiters within the input string, the corresponding array element will be blank. For example: "," would divide the string based on every occurrence of a comma. Similarly, [A_Tab, A_Space] would create a new array element every time a space or tab is encountered in the input string. OmitChars Type: String An optional list of characters (case sensitive) to exclude from the beginning and end of each array element. For example, if OmitChars is " `t", spaces and tabs will be removed from the beginning and end (but not the middle) of every element. If Delimiters is blank, OmitChars indicates which characters should be excluded from the array. MaxParts Type: Integer If omitted, it defaults to -1, which means "no limit". Otherwise, specify the maximum number of substrings to return. If non-zero, the string is split a maximum of MaxParts-1 times and the remainder of the string is returned in the last substring (excluding any leading or trailing OmitChars). Return Value Type: Array of Strings This function returns an array containing the substrings of the specified string. Remarks Whitespace characters such as spaces and tabs will be preserved unless those characters are included in the Delimiters or OmitChars parameters. Tabs and spaces can be trimmed from both ends of any variable by using Trim. For example: Var := Trim(Var) To split a string that is in standard CSV (comma separated value) format, use a parsing loop since it has built-in CSV handling. To arrange the fields in a different order prior to splitting them, use the Sort function. If you do not need the substrings to be permanently stored in memory, consider using a parsing loop -- especially if String is very large, in which case a large amount of memory would be saved. For example: Colors := "red,green,blue" Loop Parse, Colors, "," MsgBox "Color number " A_Index " is " A_LoopField Related Parsing loop, Sort, SplitPath, InStr, SubStr, StrLen, StrLower, StrUpper, StrReplace Examples Separates a sentence into an array of words and reports the fourth word. TestString := "This is a test." word_array := StrSplit(TestString, A_Space, ".") ; Omits periods. MsgBox "The 4th word is " word_array[4] Separates a comma-separated list of colors into an array of substrings and traverses them, one by one. colors := "red,green,blue" For index, color in StrSplit(colors, ",") MsgBox "Color number " index " is " color SubStr - Syntax & Usage | AutoHotkey v2 SubStr Retrieves one or more characters from the specified position in a string. NewStr := SubStr(String, StartingPos , Length) Parameters String Type: String The string whose content is copied. This may contain binary zero. StartingPos Type: Integer Specify 1 to start at the first character, 2 to start at the second, and so on (if StartingPos is 0 or beyond String's length, an empty string is returned). Specify a negative StartingPos to start at that position from the right. For example, -1 extracts the last character and -2 extracts the two last characters (but if StartingPos tries to go beyond the left end of the string, the extraction starts at the first character). Length Type: Integer If this parameter is omitted, it defaults to "all characters". Otherwise, specify the maximum number of characters to retrieve (fewer than the maximum are retrieved whenever the remaining part of the string is too short). You can also specify a negative Length to omit that many characters from the end of the returned string (an empty string is returned if all or too many characters are omitted). Return Value Type: String This function returns the requested substring of the specified string. Related RegExMatch Examples Retrieves a substring with a length of 3 characters at position 4. MsgBox SubStr("123abc789", 4, 3) ; Returns abc Retrieves a substring from the beginning and end of a string. Str := "The Quick Brown Fox Jumps Over the Lazy Dog" MsgBox SubStr(Str, 1, 19) ; Returns "The Quick Brown Fox" MsgBox SubStr(Str, -8) ; Returns "Lazy Dog" Suspend - Syntax & Usage | AutoHotkey v2 Suspend Disables or enables all or selected hotkeys and hotstrings. Suspend Mode Parameters Mode Type: Integer One of the following values: 1 or True: Suspends all hotkeys and hotstrings except those explained the Remarks section. 0 or False: Re-enables

the hotkeys and hotstrings that were disable above. -1 (default): Changes to the opposite of its previous state (On or Off). Remarks By default, the script can also be suspended via its tray icon or main window. A hotkey/hotstring can be made exempt from suspension by preceding it with the #SuspendExempt directive. An exempt hotkey/hotstring will remain enabled even while suspension is ON. This allows suspension to be turned off via a hotkey, which would otherwise be impossible since the hotkey would be suspended. The keyboard and/or mouse hooks will be installed or removed if justified by the changes made by this function. To disable selected hotkeys or hotstrings automatically based on any condition, such as the type of window that is active, use #HotIf. Suspending a script's hotkeys does not stop the script's already-running threads (if any); use Pause to do that. When a script's hotkeys are suspended, its tray icon changes to the letter S. This can be avoided by freezing the icon, which is done by specifying 1 for the last parameter of the TraySetIcon function. For example: TraySetIcon „ 1 The built-in variable A_IsSuspended contains 1 if the script is suspended and 0 otherwise. Related #SuspendExempt, Hotkeys, Hotstrings, #HotIf, Pause, ExitApp Examples Press a hotkey once to suspend all hotkeys and hotstrings. Press it again to unsuspend. #SuspendExempt ^!s::Suspend ; Ctrl+Alt+S #SuspendExempt False Sends a Suspend command to another script. DetectHiddenWindows True WM_COMMAND := 0x0111 ID_FILE_SUSPEND := 65404 PostMessage WM_COMMAND, ID_FILE_SUSPEND,,, "C:\YourScript.ahk ahk_class AutoHotkey" Switch - Syntax & Usage | AutoHotkey v2 Switch Executes one case from a list of mutually exclusive candidates. Switch SwitchValue, CaseSense { Case CaseValue1: Statements1 Case CaseValue2a, CaseValue2b: Statements2 Default: Statements3 } Parameters SwitchValue, CaseValue... Values to compare, as described in the remarks below. CaseSense Type: Integer or String Optionally specify one of the following values to force all values to be compared as strings: "On" or 1 (True): Each comparison is case sensitive. "Off" or 0 (False): The letters A-Z are considered identical to their lowercase counterparts. "Locale": Each comparison is case insensitive according to the rules of the current user's locale. For details, see StrCompare. Remarks If present, SwitchValue is evaluated once and compared to each case value until a match is found, and then that case is executed. Otherwise, the first case which evaluates to true (non-zero and non-empty) is executed. If there is no matching case and a Default is present, it is executed. As with the = and == operators, when CaseSense is omitted, numeric comparison is performed if SwitchValue and the case value are both pure numbers, or if one is a pure number and the other is a numeric string. Each case value is considered separately and does not affect the type of comparison used for other case values. If the CaseSense parameter is present, all values are compared as strings, not as numbers, and a TypeError is thrown if SwitchValue or a CaseValue evaluates to an object. If the CaseSense parameter is omitted, string comparisons are case-sensitive by default. Each case may list up to 20 values. Each value must be an expression, but can be a simple one such as a literal number, quoted string or variable. Case and Default must be terminated with a colon :. The first statement of each case may be below Case or on the same line, following the colon. Each case implicitly ends at the next Case/Default or the closing brace. Unlike the switch statement found in some other languages, there is no implicit fall-through and Break is not used (except to break out of an enclosing loop). As all cases are enclosed in the same block, a label defined in one case can be the target of Goto from another case. However, if a label is placed immediately above Case or Default, it targets the end of the previous case, not the beginning of the next one. Default is not required to be listed last. Related If, Else, Blocks Examples To test this example, type [followed by one of the abbreviations listed below, any other 5 characters, or Enter/Esc/Tab/.; or wait for 4 seconds. ~[:: { Hook := InputHook("V T5 L4 C", "{enter}.{esc}{tab}", "btw,otoh,fl,ahk,ca") Hook.Start() Hook.Wait() switch Hook.EndReason { case "Max": MsgBox 'You entered "' Hook.Input '", which is the maximum length of text.' case "Timeout": MsgBox 'You entered "' Hook.Input '" at which time the input timed out.' case "EndKey": MsgBox 'You entered "' Hook.Input '" and terminated the input with ' Hook.EndReason '.' default: ; Match switch Hook.Input { case "btw": Send "{backspace 3}by the way" case "otoh": Send "{backspace 4}on the other hand" case "fl": Send "{backspace 2}Florida" case "ca": Send "{backspace 2}California" case "ahk": Send "{backspace 3}" Run "https://www.autohotkey.com" } } } SysGet - Syntax & Usage | AutoHotkey v2 SysGet Retrieves dimensions of system objects, and other system properties. Value := SysGet(Property) Parameters Property Type: Integer Specify one of the numbers from the tables below to retrieve the corresponding value. Return Value Type: Integer This function returns the value of the specified system property. Commonly Used Number Description 80 SM_CMONITORS: Number of display monitors on the desktop (not including "non-display pseudo-monitors"). 43 SM_CMOUSEBUTTONS: Number of buttons on mouse (0 if no mouse is installed). 16, 17 SM_CXFULLSCREEN, SM_CYFULLSCREEN: Width and height of the client area for a full-screen window on the primary display monitor, in pixels. 61, 62 SM_CXMAXIMIZED, SM_CYMAXIMIZED: Default dimensions, in pixels, of a maximized top-level window on the primary display monitor. 59, 60 SM_CXMAXTRACK, SM_CYMAXTRACK: Default maximum dimensions of a window that has a caption and sizing borders, in pixels. This metric refers to the entire desktop. The user cannot drag the window frame to a size larger than these dimensions. 28, 29 SM_CXMIN, SM_CYMIN: Minimum width and height of a window, in pixels. 57, 58 SM_CXMINIMIZED, SM_CYMINIMIZED: Dimensions of a minimized window, in pixels. 34, 35 SM_CXMINTRACK, SM_CYMINTRACK: Minimum tracking width and height of a window, in pixels. The user cannot drag the window frame to a size smaller than these dimensions. A window can override these values by processing the WM_GETMINMAXINFO message. 0, 1 SM_CXSCREEN, SM_CYSCREEN: Width and height of the screen of the primary display monitor, in pixels. These are the same as the built-in variables A_ScreenWidth and A_ScreenHeight. 78, 79 SM_CXVIRTUALSCREEN, SM_CYVIRTUALSCREEN: Width and height of the virtual screen, in pixels. The virtual screen is the bounding rectangle of all display monitors. The SM_XVIRTUALSCREEN, SM_YVIRTUALSCREEN metrics are the coordinates of the top-left corner of the virtual screen. 19 SM_MOUSEPRESENT: Nonzero if a mouse is installed; zero otherwise. 75 SM_MOUSEWHEELPRESENT: Nonzero if a mouse with a wheel is installed; zero otherwise. 63 SM_NETWORK: Least significant bit is set if a network is present; otherwise, it is cleared. The other bits are reserved for future use. 8193 SM_REMOTECONTROL: This system metric is used in a Terminal Services environment. Its value is nonzero if the current session is remotely controlled; zero otherwise. 4096 SM_REMOTESESSION: This system metric is used in a Terminal Services environment. If the calling process is associated with a Terminal Services client session, the return value is nonzero. If the calling process is associated with the Terminal Server console session, the return value is zero. The console session is not necessarily the physical console. 70 SM_SHOWSOUNDS: Nonzero if the user requires an application to present information visually in situations where it would otherwise present the information only in audible form; zero otherwise. 8192 SM_SHUTTINGDOWN: Nonzero if the current session is shutting down; zero otherwise. Windows 2000: The retrieved value is always 0. 23 SM_SWAPBUTTON: Nonzero if the meanings of the left and right mouse buttons are swapped; zero otherwise. 76, 77 SM_XVIRTUALSCREEN, SM_YVIRTUALSCREEN: Coordinates for the left side and the top of the virtual screen. The virtual screen is the bounding rectangle of all display monitors. By contrast, the SM_CXVIRTUALSCREEN, SM_CYVIRTUALSCREEN metrics (further above) are the width and height of the virtual screen. Not Commonly Used Number Description 56 SM_ARRANGE: Flags specifying how the system arranged minimized windows. See Microsoft Docs for more information. 67 SM_CLEANBOOT: Specifies how the system was started: 0 = Normal boot 1 = Fail-safe boot 2 = Fail-safe with network boot 5, 6 SM_CXBORDER, SM_CYBORDER: Width and height of a window border, in pixels. This is equivalent to the SM_CXEDGE value for windows with the 3-D look. 13, 14 SM_CXCURSOR, SM_CYCURSOR: Width and height of a cursor, in pixels. The system cannot create cursors of other sizes. 36, 37 SM_CXDOUBLECLK, SM_CYDOUBLECLK: Width and height of the rectangle around the location of a first click in a double-click sequence, in pixels. The second click must occur within this rectangle for the system to consider the two clicks a double-click. (The two clicks must also occur within a specified time.) 68, 69 SM_CXDRAG, SM_CYDRAG: Width and height of a rectangle centered on a drag point to allow for limited movement of the mouse pointer before a drag operation begins. These values are in pixels. It allows the user to click and release the mouse button easily without unintentionally starting a drag operation. 45, 46 SM_CXEDGE, SM_CYEDGE: Dimensions of a 3-D border, in pixels. These are the 3-D counterparts of SM_CXBORDER and SM_CYBORDER. 7, 8 SM_CXFIXEDFRAME, SM_CYFIXEDFRAME (synonymous with SM_CXDLGFRAME, SM_CYDLGFRAME): Thickness of the frame around the perimeter of a window that has a caption but is not sizable, in pixels. SM_CXFIXEDFRAME is the height of the horizontal border and SM_CYFIXEDFRAME is the width of the vertical border. 83, 84 SM_CXFOCUSBORDER, SM_CYFOCUSBORDER: Width (in pixels) of the left and right edges and the height of the top and bottom edges of a control's focus rectangle. Windows 2000: The retrieved value is always 0. 21, 3 SM_CXHSCROLL, SM_CYHSCROLL: Width of the arrow bitmap on a horizontal scroll bar, in pixels; and height of a horizontal scroll bar, in pixels. 10 SM_CXHTHUMB: Width of the thumb box in a horizontal scroll bar, in pixels. 11, 12 SM_CXICON, SM_CYICON: Default width and height of an icon, in pixels. 38, 39 SM_CXICONSPACING, SM_CYICONSPACING: Dimensions of a grid cell for items in large icon view, in pixels. Each item fits into a rectangle of this size when arranged. These values are always greater than or equal to SM_CXICON and SM_CYICON. 71, 72 SM_CXMENUCHECK, SM_CYMENUCHECK: Dimensions of the default menu check-mark bitmap, in pixels. 54, 55 SM_CXMENUSIZE, SM_CYMENUSIZE: Dimensions of menu bar buttons, such as the child window close button used in the multiple document interface, in pixels. 47, 48 SM_CXMINSPACING SM_CYMINSPACING: Dimensions of a grid cell for a minimized window, in pixels. Each minimized window fits into a rectangle this size when arranged. These values are always greater than or equal to SM_CXMINIMIZED and SM_CYMINIMIZED. 30, 31 SM_CXSIZE, SM_CYSIZE: Width and height of a button in a window's caption or title bar, in pixels. 32, 33 SM_CXSIZEFRAME, SM_CYSIZEFRAME: Thickness of the sizing border around the perimeter of a window that can be resized, in pixels. SM_CXSIZEFRAME is the width of the horizontal border, and SM_CYSIZEFRAME is the height of the vertical border. Synonymous with SM_CXFRAME and SM_CYFRAME. 49, 50 SM_CXSMICON, SM_CYSMICON: Recommended dimensions of a small icon, in pixels. Small icons typically appear in window captions and in small icon view. 52, 53 SM_CXSMSIZE SM_CYSMSIZE: Dimensions of small caption buttons, in pixels. 2, 20 SM_CXVSCROLL, SM_CYVSCROLL: Width of a vertical scroll bar, in pixels; and height of the arrow bitmap on a vertical scroll bar, in pixels. 4 SM_CYCAPTION: Height of a caption area, in pixels. 18 SM_CYKANJIWINDOW: For double byte character set versions of the system, this is the height of the Kanji window at the bottom of the screen, in pixels. 15 SM_CYMENU: Height of a single-line menu bar, in pixels. 51 SM_CYSMCAPTION: Height of a small caption, in pixels. 9 SM_CYVTHUMB: Height of the thumb box in a vertical scroll bar, in pixels. 42 SM_DBCSENABLED: Nonzero if User32.dll supports DBCS; zero otherwise. 22 SM_DEBUG: Nonzero if the debug version of User.exe is installed; zero otherwise. 82 SM_IMMENABLED: Nonzero if Input Method Manager/Input Method Editor features are enabled; zero otherwise. SM_IMMENABLED indicates whether the system is ready to use a Unicode-based IME on a

Unicode application. To ensure that a language-dependent IME works, check SM_DBCSENABLED and the system ANSI code page. Otherwise the ANSI-to-Unicode conversion may not be performed correctly, or some components like fonts or registry setting may not be present. 40 SM_MENUDROPALIGNMENT: Nonzero if drop-down menus are right-aligned with the corresponding menu-bar item; zero if the menus are left-aligned. 74 SM_MIDEASTENABLED: Nonzero if the system is enabled for Hebrew and Arabic languages, zero if not. 41 SM_PENWINDOWS: Nonzero if the Microsoft Windows for Pen computing extensions are installed; zero otherwise. 44 SM_SECURE: Nonzero if security is present; zero otherwise. 81 SM_SAMEDISPLAYFORMAT: Nonzero if all the display monitors have the same color format, zero otherwise. Note that two displays can have the same bit depth, but different color formats. For example, the red, green, and blue pixels can be encoded with different numbers of bits, or those bits can be located in different places in a pixel's color value. Remarks SysGet simply calls the GetSystemMetrics function, which may support more system properties than are documented here. Related DllCall, Win functions, Monitor functions Examples Retrieves the number of mouse buttons and stores it in MouseButtonCount. MouseButtonCount := SysGet(43) Retrieves the width and height of the virtual screen and stores them in VirtualScreenWidth and VirtualScreenHeight. VirtualScreenWidth := SysGet(78) VirtualScreenHeight := SysGet(79) SysGetIPAddresses - Syntax & Usage | AutoHotkey v2 SysGetIPAddresses Returns an array of the system's IPv4 addresses. Addresses := SysGetIPAddresses() Parameters This function has no parameters. Return Value Type: Array of Strings This function returns an array, where each element is an IPv4 address string such as "192.168.0.1". Remarks Currently only IPv4 is supported. This function returns only the IP addresses of the computer's network adapters. If the computer is connected to the Internet through a router, this will not include the computer's public (Internet) IP address. To determine the computer's public IP address, use an external web API. For example: whr := ComObject("WinHttp.WinHttpRequest.5.1") whr.Open("GET", "https://api.ipify.org") whr.Send() MsgBox "Public IP address: " whr.ResponseText Related A_ComputerName Examples Retrieves and reports the system's IPv4 addresses. addresses := SysGetIPAddresses() msg := "IP addresses:`n" for n, address in addresses msg .= address "`n" MsgBox msg Thread - Syntax & Usage | AutoHotkey v2 Thread Sets the priority or interruptibility of threads. It can also temporarily disable all timers. Thread SubFunction , Value1, Value2 The SubFunction, Value1, and Value2 parameters are dependent upon each other and their usage is described below. Sub-functions For SubFunction, specify one of the following: NoTimers: Prevents interruptions from any timers. Priority: Changes the priority level of the current thread. Interrupt: Changes the duration of interruptibility for newly launched threads. NoTimers Prevents interruptions from any timers. Thread "NoTimers" , TrueOrFalse This sub-function prevents interruptions from any timers until the current thread either ends, executes Thread "NoTimers", false, or is interrupted by another thread that allows timers (in which case timers can interrupt the interrupting thread until it finishes). If this setting is not changed by the auto-execute thread, all threads start off as interruptible by timers (though the settings of the Interrupt sub-function [below] will still apply). By contrast, if the auto-execute thread turns on this setting but never turns it off, every newly launched thread (such as a hotkey, custom menu item, or timer) starts off immune to interruptions by timers. Regardless of the default setting, timers will always operate when the script has no threads (unless Pause has been turned on). Thread "NoTimers" is equivalent to Thread "NoTimers", true. In addition, since TrueOrFalse is an expression, true resolves to 1, and false to 0. Priority Changes the priority level of the current thread. Thread "Priority", Level Specify for Level an integer between -2147483648 and 2147483647 (or an expression) to indicate the current thread's new priority. This has no effect on other threads. See Threads for details. Due to its ability to buffer events, the function Critical is generally superior to this sub-function. On a related note, the OS's priority level for the entire script can be changed via ProcessSetPriority. For example: ProcessSetPriority "High" Interrupt Changes the duration of interruptibility for newly launched threads. Thread "Interrupt" , Duration, LineCount Note: This sub-function should be used sparingly because most scripts perform more consistently with settings close to the defaults. By default, every newly launched thread is uninterruptible for a Duration of 15 milliseconds or a LineCount of 1000 script lines, whichever comes first. This gives the thread a chance to finish rather than being immediately interrupted by another thread that is waiting to launch (such as a buffered hotkey or a series of timed subroutines that are all due to be run). Note: Any Duration less than 17 might result in a shorter actual duration or immediate interruption, since the system tick count has a minimum resolution of 10 to 16 milliseconds. However, at least one line will execute before the thread becomes interruptible, allowing the script to enable Critical, if needed. If either parameter is 0, each newly launched thread is immediately interruptible. If either parameter is -1, the thread cannot be interrupted as a result of that parameter. The maximum for both parameters is 2147483647. This setting is global, meaning that it affects all subsequent threads, even if this function was not called by the auto-execute thread. However, interrupted threads are unaffected because their period of uninterruptibility has already expired. Similarly, the current thread is unaffected except if it is uninterruptible at the time the LineCount parameter is changed, in which case the new LineCount will be in effect for it. If a hotkey is pressed or a custom menu item is selected while the current thread is uninterruptible, that event will be buffered. In other words, it will launch when the current thread finishes or becomes interruptible, whichever comes first. The exception to this is when the current thread becomes interruptible before it finishes, and it is of higher priority than the buffered event; in this case the buffered event is unbuffered and discarded. Regardless of this sub-function, a thread will become interruptible the moment it displays a MsgBox, InputBox, FileSelect, or DirSelect dialog. Either parameter can be left blank to avoid changing it. Remarks Due to its greater flexibility and its ability to buffer events, the function Critical is generally more useful than Thread "Interrupt" and Thread "Priority". Related Critical, Threads, Hotkey, Menu object, SetTimer, Process functions Examples Makes the priority of the current thread slightly above average. Thread "Priority", 1 Makes each newly launched thread immediately interruptible. Thread "Interrupt", 0 Makes each thread interruptible after 50 ms or 2000 lines, whichever comes first. Thread "Interrupt", 50, 2000 Throw - Syntax & Usage | AutoHotkey v2 Throw Signals the occurrence of an error. This signal can be caught by a try-catch statement. Throw Value Parameters Value A value to throw; typically an Error object. For example: throw ValueError("Parameter #1 invalid", -1, theBadParam) Values of all kinds can be thrown, but if Catch is used without specifying a class (or Try is used without Catch or Finally), it will only catch instances of the Error class. While execution is within a Catch, Value can be omitted to re-throw the caught value (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 line with throw does not need to be physically contained by the catch statement's body; it can be used by a called function. Remarks The space or tab after throw is optional if the expression is enclosed in parentheses, as in throw(Error()). A thrown value or runtime error can be caught by Try-Catch. In such cases, execution is transferred into the catch statement or to the next statement after the try. If a thrown value is not caught, the following occurs: Any active OnError callbacks are called. Each callback may inspect Value and either suppress or allow further callbacks and default handling. By default, an error message is displayed based on what was thrown. If Value is an Object and owns a value property named Message, its value is used as the message. If Value is a non-numeric string, it is used as the message. In any other case, a default message is used. If Value is numeric, it is shown below the default message. The thread exits. Note that this does not necessarily occur for continuable errors, but throw is never continuable. Related Error Object, Try, Catch, Finally, OnError Examples See Try. ToolTip - Syntax & Usage | AutoHotkey v2 ToolTip Creates an always-on-top window anywhere on the screen. ToolTip Text, X, Y, WhichToolTip Parameters Text Type: String If blank or omitted, the existing tooltip (if any) will be hidden. Otherwise, this parameter is the text to display in the tooltip. To create a multi-line tooltip, use the linefeed character (`n) in between each line, e.g. Line1`nLine2. If Text is long, it can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. X, Y Type: Integer The X and Y position of the tooltip relative to the active window's client area (use CoordMode "ToolTip" to change to screen coordinates). If the coordinates are omitted, the tooltip will be shown near the mouse cursor. WhichToolTip Type: Integer Omit this parameter if you don't need multiple tooltips to appear simultaneously. Otherwise, this is a number between 1 and 20 to indicate which tooltip window to operate upon. If unspecified, that number is 1 (the first). Return Value Type: Integer or empty string If a tooltip is being shown or updated, this function returns the tooltip window's unique ID (HWND), which can be used to move the tooltip or send Tooltip Control Messages. If Text is omitted or blank, the return value is an empty string. Remarks A tooltip usually looks like this: If the X & Y coordinates caused the tooltip to run off-screen, or outside the monitor's working area on Windows 8 or later, it is repositioned to be entirely visible. The tooltip is displayed until one of the following occurs: The script terminates. The ToolTip function is executed again with a blank Text parameter. The user clicks on the tooltip (this behavior may vary depending on operating system version). A GUI window may be made the owner of a tooltip by means of the OwnDialogs option. Such a tooltip is automatically destroyed when its owner is destroyed. Related CoordMode, TrayTip, GUI, MsgBox, InputBox, FileSelect, DirSelect Examples Shows a multiline tooltip at a specific position in the active window. ToolTip "Multiline`nTooltip", 100, 150 Hides a tooltip after a certain amount of time without having to use Sleep (which would stop the current thread). ToolTip "Timed ToolTip`nThis will be displayed for 5 seconds." SetTimer () => ToolTip(), -5000 TraySetIcon - Syntax & Usage | AutoHotkey v2 TraySetIcon Changes the script's tray icon (which is also used by GUI and dialog windows). TraySetIcon(FileName, IconNumber, Freeze) Parameters FileName Type: String The path of an icon or image file. For a list of supported formats, see the Picture control. Specify an asterisk (*) to restore the script to its default icon. IconNumber Type: Integer To use an icon group other than the first one in the file, specify its number for IconNumber (if omitted, it defaults to 1). For example, 2 would load the default icon from the second icon group. If IconNumber is negative, its absolute value is assumed to be the resource ID of an icon within an executable file. Freeze Type: Boolean Specify 1 (true) to freeze the icon, or 0 (false) to unfreeze it (or omit it to keep the frozen/unfrozen state unchanged). When the icon has been frozen, Pause and Suspend will not change it. Note: To freeze or unfreeze the current icon, use 1 (true) or 0 (false) as in the following example: TraySetIcon(,, 1). Remarks Changing the tray icon also changes the icon displayed by InputBox and subsequently-created GUI windows. Compiled scripts are also affected even if a custom icon was specified at the time of compiling. Note: Changing the icon will not unhide the tray icon if it was previously hidden by means such as #NoTrayIcon; to do that, use A_IconHidden := false. Slight distortion may occur when loading tray icons from file types other than .ICO. This is especially true for 16x16 icons. To prevent this, store the desired tray icon inside a .ICO file. There are some icons built into the operating system's DLLs and CPLs that might be useful. For example: TraySetIcon "Shell32.dll", 174. A bitmap or icon handle can be used instead of a filename. For example, TraySetIcon "HBITMAP:" handle. The built-in variables A_IconNumber and A_IconFile contain the number and name (with full path) of the current icon (both are blank if the icon is the default). The tray icon's tooltip can be changed by assigning a value to A_IconTip. Related

#NoTrayIcon, TrayTip, Menu object TrayTip - Syntax & Usage | AutoHotkey v2 TrayTip Creates a balloon message window near the tray icon. On Windows 10, a toast notification may be shown instead. TrayTip Text, Title, Options Parameters Text Type: String The message to display. Only the first 265 characters will be displayed. Carriage return (`r) or linefeed (`n) may be used to create multiple lines of text. For example: Line1`nLine2. If Text is long, it can be broken up into several shorter lines by means of a continuation section, which might improve readability and maintainability. It is possible to show a window with only a title by leaving Text blank. Title Type: String The title of the window. Only the first 73 characters will be displayed. If Title is blank, the title line will be entirely omitted from the window, making it vertically shorter. Options Type: String or Integer Either an integer value (a combination by addition or bitwise-OR) or a string of zero or more case-insensitive options separated by at least one space or tab. One or more numeric options may also be included in the string. Function Dec Hex String Info icon 1 0x1 Iconi Warning icon 2 0x2 Icon! Error icon 3 0x3 Iconx Tray icon 4 0x4 N/A Do not play the notification sound. 16 0x10 Mute Use the large version of the icon. 32 0x20 N/A If omitted, it defaults to 0, which is no icon. The icon is also not shown by the balloon window if it lacks a Title (this does not apply to Windows 10 toast notifications). On Windows 10, the small tray icon is generally displayed even if the "tray icon" option (4) is omitted, and specifying this option may cause the program's name to be shown in the notification. To Hide the Window To hide a TrayTip balloon window, omit both Text and Title. For example: TrayTip To hide a Windows 10 toast notification, temporarily remove the tray icon. For example: TrayTip "#1", "This is TrayTip #1" Sleep 3000 ; Let it display for 3 seconds. HideTrayTip TrayTip "#2", "This is the second notification." Sleep 3000 ; Copy this function into your script to use it. HideTrayTip() { TrayTip ; Attempt to hide it the normal way. if SubStr(A_OSVersion,1,3) = "10." { A_IconHidden := true Sleep 200 ; It may be necessary to adjust this sleep. A_IconHidden := false } } This may not always work, according to at least one report. Remarks On Windows 10, a TrayTip window usually looks like this: Windows 10 replaces all balloon windows with toast notifications by default (this can be overridden via group policy). Calling TrayTip multiple times will usually cause multiple notifications to be placed in a "queue" instead of each notification replacing the last. TrayTip has no effect if the script lacks a tray icon (via #NoTrayIcon or A_IconHidden := true). TrayTip also has no effect if the following REG_DWORD value exists and has been set to 0: HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced >> EnableBalloonTips On a related note, there is a tooltip displayed whenever the user hovers the mouse over the script's tray icon. The contents of this tooltip can be changed via: A_IconTip := "My New Text". Related ToolTip, SetTimer, Menu object, MsgBox, InputBox, FileSelect, DirSelect Examples Shows a multiline balloon message or toast notification for 20 seconds near the tray icon without playing the notification sound. It also has a title and contains an info icon. TrayTip "Multiline`nText", "My Title", "Iconi Mute" Provides a more precise control over the display time without having to use Sleep (which would stop the current thread). For Windows 10, replace the HideTrayTip function definition with the one defined above. TrayTip "This will be displayed for 5 seconds.", "Timed TrayTip" SetTimer HideTrayTip, -5000 HideTrayTip() { TrayTip } Permanently displays a TrayTip by refreshing it periodically via timer. Note that this probably won't work well on Windows 10 for reasons described above. SetTimer RefreshTrayTip, 1000 RefreshTrayTip ; Call it once to get it started right away. RefreshTrayTip() { TrayTip "This is a more permanent TrayTip.", "Refreshed TrayTip", "Mute" } TreeView (GUI) - Syntax & Usage | AutoHotkey v2 TreeView Table of Contents Introduction and Simple Example Options and Styles for the Options Parameter Built-in Methods for TreeViews Events Remarks Examples Introduction and Simple Example A Tree-View displays a hierarchy of items by indenting child items beneath their parents. The most common example is Explorer's tree of drives and folders. A TreeView usually looks like this: The syntax for creating a TreeView is: TV := GuiObj.Add("TreeView", Options) Or: TV := GuiObj.AddTreeView(Options) Here is a working script that creates and displays a simple hierarchy of items: MyGui := Gui() TV := MyGui.Add("TreeView") P1 := TV.Add("First parent") P1C1 := TV.Add("Parent 1's first child", P1) ; Specify P1 to be this item's parent. P2 := TV.Add("Second parent") P2C1 := TV.Add("Parent 2's first child", P2) P2C2 := TV.Add("Parent 2's second child", P2) P2C2C1 := TV.Add("Child 2's first child", P2C2) MyGui.Show ; Show the window and its TreeView. Options and Styles for the Options Parameter Background: Specify the word Background followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: BackgroundSilver, BackgroundFFDD99. If this option is not present, the TreeView initially defaults to the system's default background color. Specifying BackgroundDefault or -Background applies the system's default background color (usually white). For example, a TreeView can be restored to the default color via TV.Opt("+BackgroundDefault"). Buttons: Specify -Buttons (minus Buttons) to avoid displaying a plus or minus button to the left of each item that has children. C: Text color. Specify the letter C followed immediately by a color name (see color chart) or RGB value (the 0x prefix is optional). Examples: cRed, cFF2211, c0xFF2211, cDefault. Checked: Provides a checkbox at the left side of each item. When adding an item, specify the word Check in its options to have the box to start off checked instead of unchecked. The user may either click the checkbox or press the spacebar to check or uncheck an item. To discover which items in a TreeView are currently checked, call the GetNext method or Get method. HScroll: Specify -HScroll (minus HScroll) to disable horizontal scrolling in the control (in addition, the control will not display any horizontal scroll bar). ImageList: This is the means by which icons are added to a TreeView. Specify the word ImageList followed immediately by the ImageListID returned from a previous call to IL_Create. This option has an effect only when creating a TreeView (however, the SetImageList method does not have this limitation). Here is a working example: MyGui := Gui() ImageListID := IL_Create(10) ; Create an ImageList with initial capacity for 10 icons. Loop 10 ; Load the ImageList with some standard system icons. IL_Add(ImageListID, "shell32.dll", A_Index) TV := MyGui.Add("TreeView", "ImageList" . ImageListID) TV.Add("Name of Item", 0, "Icon4") ; Add an item to the TreeView and give it a folder icon. MyGui.Show Lines: Specify -Lines (minus Lines) to avoid displaying a network of lines connecting parent items to their children. However, removing these lines also prevents the plus/minus buttons from being shown for top-level items. ReadOnly: Specify -ReadOnly (minus ReadOnly) to allow editing of the text/name of each item. To edit an item, select it then press F2 (see the WantF2 option below). Alternatively, you can click an item once to select it, wait at least half a second, then click the same item again to edit it. After being edited, an item can be alphabetically repositioned among its siblings via the following example: TV := MyGui.Add("TreeView", "-ReadOnly") TV.OnEvent("ItemEdit", MyTree_Edit) ; Call MyTree_Edit whenever a user has finished editing an item. ; ... MyTree_Edit(TV, Item) { TV.Modify(TV.GetParent(Item), "Sort") ; This works even if the item has no parent. } R: Rows of height (upon creation). Specify the letter R followed immediately by the number of rows for which to make room inside the control. For example, R10 would make the control 10 items tall. WantF2: Specify -WantF2 (minus WantF2) to prevent F2 from editing the currently selected item. This setting is ignored unless -ReadOnly is also in effect. (Unnamed numeric styles): Since styles other than the above are rarely used, they do not have names. See the TreeView styles table for a list. Built-in Methods for TreeViews In addition to the default methods/properties of a GUI control, TreeView controls have the following methods (defined in the Gui.TreeView class). Item methods: Add: Adds a new item to the TreeView. Modify: Modifies the attributes and/or name of an item. Delete: Deletes the specified item or all items. Retrieval methods: GetSelection: Returns the selected item's ID number. GetCount: Returns the total number of items in the control. GetParent: Returns the specified item's parent as an item ID. GetChild: Returns the ID number of the specified item's first/top child. GetPrev: Returns the ID number of the sibling above the specified item. GetNext: Returns the ID number of the next item below the specified item. GetText: Retrieves the text/name of the specified item. Get: Returns a non-zero value (item ID) if the specified item has the specified attribute. Other methods: SetImageList: Sets or replaces an ImageList for displaying icons. Add Adds a new item to the TreeView and returns its unique Item ID number (or 0 upon failure). UniqueID := TV.Add(Name, ParentItemID, Options) Name Type: String The displayed text of the item, which can be text or numeric (including numeric expression results). ParentItemID Type: Integer The ID number of the new item's parent (omit it or specify 0 to add the item at the top level). Options Type: String Contains zero or more words from the list below (not case sensitive). Separate each word from the next with a space or tab. To remove an option, precede it with a minus sign. To add an option, a plus sign is permitted but not required. Bold: Displays the item's name in a bold font. To later un-bold the item, use TV.Modify(ItemID, "-Bold"). The word Bold may optionally be followed immediately by a 0 or 1 to indicate the starting state. Check: Shows a checkmark to the left of the item (if the TreeView has checkboxes). To later uncheck it, use TV.Modify(ItemID, "-Check"). The word Check may optionally be followed immediately by a 0 or 1 to indicate the starting state. In other words, both "Check" and "Check" . VarContainingOne are the same (the period used here is the concatenation operator). Expand: Expands the item to reveal its children (if any). To later collapse the item, use TV.Modify(ItemID, "-Expand"). If there are no children, the Modify method returns 0 instead of the item's ID. By contrast, the Add method marks the item as expanded in case children are added to it later. Unlike "Select" (below), expanding an item does not automatically expand its parent. Finally, the word Expand may optionally be followed immediately by a 0 or 1 to indicate the starting state. In other words, both "Expand" and "Expand" . VarContainingOne are the same. First | Sort | N: These options apply only to the Add method. They specify the new item's position relative to its siblings (a sibling is any other item on the same level). If none of these options is present, the new item is added as the last/bottom sibling. Otherwise, specify First to add the item as the first/top sibling, or specify Sort to insert it among its siblings in alphabetical order. If a plain integer (N) is specified, it is assumed to be ID number of the sibling after which to insert the new item (if integer N is the only option present, it does not have to be enclosed in quotes). Icon: Specify the word Icon followed immediately by the number of this item's icon, which is displayed to the left of the item's name. If this option is absent, the first icon in the ImageList is used. To display a blank icon, specify a number that is larger than the number of icons in the ImageList. If the control lacks an ImageList, no icon is displayed nor is any space reserved for one. Select: Selects the item. Since only one item at a time can be selected, any previously selected item is automatically de-selected. In addition, this option reveals the newly selected item by expanding its parent(s), if necessary. To find out the current selection, call the GetSelection method. Sort: For the Modify method, this option alphabetically sorts the children of the specified item. To instead sort all top-level items, use TV.Modify(0, "Sort"). If there are no children, 0 is returned instead of the ID of the modified item. Vis: Ensures that the item is completely visible by scrolling the TreeView and/or expanding its parent, if necessary. VisFirst: Same as above except that the TreeView is also scrolled so that the item appears at the top, if possible. This option is typically more effective when used with the Modify method than with the Add method. Note: When adding a large number of items, performance can be improved by using TV.Opt("-Redraw") before adding the items and TV.Opt("+Redraw") afterward. See Redraw for more details. Modify Modifies the attributes and/or name of an item, and returns the item's own ID. CurrentItemID := TV.Modify(ItemID , Options, NewName) ItemID Type: Integer The item to modify. When only this parameter is

present, the specified item is selected. Options Type: String See the list above. NewName Type: String The new name of the item. If omitted, the current name is left unchanged. Delete Deletes the specified item or all items. TV.Delete(ItemID) ItemID Type: Integer The item to delete. If omitted, all items in the TreeView are deleted. GetSelection Returns the selected item's ID number. SelectedItemID := TV.GetSelection() GetCount Returns the total number of items in the control. CountNumber := TV.GetCount() Note: The return value will be retrieved instantly, because the control keeps track of the count. GetParent Returns the specified item's parent as an item ID. ParentItemID := TV.GetParent(ItemID) ItemID Type: Integer The item to check. Items at the top level have no parent and thus return 0. GetChild Returns the ID number of the specified item's first/top child (or 0 if none). TopChildID := TV.GetChild(ParentItemID) ParentItemID Type: Integer The parent item to check. If this parameter is 0, the ID number of the first/top item in the TreeView is returned. GetPrev Returns the ID number of the sibling above the specified item (or 0 if none). PrevItemID := TV.GetPrev(ItemID) ItemID Type: Integer The item to check. GetNext Returns the ID number of the next item below the specified item (or 0 if none). NextItemID := TV.GetNext(ItemID, ItemType) ItemID Type: Integer The item to check. If this parameter is 0 or omitted, the ID number of the first/top item in the TreeView is returned. ItemType Type: String If omitted, the ID number of the sibling below the specified item will be retrieved. Otherwise specify "Full" or "F" to retrieve the next item regardless of its relationship to the specified item; or specify "Check", "Checked", or "C" to get only the next item with a checkmark. The following example traverse the entire tree, item by item: ItemID := 0 ; Causes the loop's first iteration to start the search at the top of the tree. Loop { ItemID := TV.GetNext(ItemID, "Full") ; Replace "Full" with "Checked" to find all checkmarked items. if not ItemID ; No more items in tree. break ItemText := TV.GetText(ItemID) MsgBox('The next Item is ' ItemID ', whose text is "' ItemText '".') } GetText Retrieves the text/name of the specified item. RetrievedText := TV.GetText(ItemID) ItemID Type: Integer The ID number of the item, whose text to be retrieved. RetrievedText has a maximum length of 8191. Get Returns a non-zero value (item ID) if the specified item has the specified attribute. CurrentItemID := TV.Get(ItemID, Attribute) ItemID Type: Integer The item to check. Attribute Type: String Specify "E", "Expand", or "Expanded" to determine if the item is currently expanded (that is, its children are being displayed); specify "C", "Check", or "Checked" to determine if the item has a checkmark; or specify "B" or "Bold" to determine if the item is currently bold in font. Tip: Since an IF-statement sees any non-zero value as "true", if TV.Get(ItemID, "Checked") = ItemID and if TV.Get(ItemID, "Checked") are functionally identical. SetImageList Sets or replaces an ImageList for displaying icons, and returns the ImageListID that was previously associated with this control (or 0 if none). PrevImageListID := TV.SetImageList(ImageListID , IconType) ImageListID Type: Integer The number returned from a previous call to IL_Create. IconType Type: Integer If omitted, it defaults to 0. Otherwise, specify 2 for state icons (state icons are not yet directly supported, but they could be used via SendMessage). Any such detached ImageList should normally be destroyed via IL_Destroy. Events The following events can be detected by calling OnEvent to register a callback function or method: EventRaised when... ClickThe control is clicked. DoubleClickThe control is double-clicked. ContextMenuThe user right-clicks the control or presses Menu or Shift+F10 while the control has the keyboard focus. FocusThe control gains the keyboard focus. LoseFocusThe control loses the keyboard focus. ItemCheckAn item is checked or unchecked. ItemEditAn item's label is edited by the user. ItemExpandAn item is expanded or collapsed. ItemSelectAn item is selected. Additional (rarely-used) notifications can be detected by using OnNotify. These notifications are documented at Microsoft Docs. Microsoft Docs does not show the numeric value of each notification code; those can be found in the Windows SDK or by searching the Internet. Remarks To detect when the user has pressed Enter while a TreeView has focus, use a default button (which can be hidden if desired). For example: MyGui.Add("Button", "Hidden Default", "OK").OnEvent("Click", ButtonOK) ... ButtonOK(*) { global if MyGui.FocusedCtrl != TV return MsgBox("Enter was pressed. The selected item ID is " TV.GetSelection()) } In addition to navigating from item to item with the keyboard, the user may also perform incremental search by typing the first few characters of an item's name. This causes the selection to jump to the nearest matching item. Although any length of text can be stored in each item of a TreeView, only the first 260 characters are displayed. Although the theoretical maximum number of items in a TreeView is 65536, item-adding performance will noticeably decrease long before then. This can be alleviated somewhat by using the redraw tip described in the Add method. Unlike ListViews, a TreeView's ImageList is not automatically destroyed when the TreeView is destroyed. Therefore, a script should call IL_Destroy after destroying a TreeView's window if the ImageList will not be used for anything else. However, this is not necessary if the script will soon be exiting because all ImageLists are automatically destroyed at that time. A script may create more than one TreeView per window. To perform actions such as resizing, hiding, or changing the font of a TreeView, see GuiControl object. Tree View eXtension (TVX) extends TreeViews to support moving, inserting and deleting. It is demonstrated at www.autohotkey.com/forum/topic19021.html Related ListView, Other Control Types, Gui(), ContextMenu event, Gui object, GuiControl object, TreeView styles table Examples The following is a working script that is more elaborate than the one near the top of this page. It creates and displays a TreeView containing all folders in the all-users Start Menu. When the user selects a folder, its contents are shown in a ListView to the right (like Windows Explorer). In addition, a StatusBar control shows information about the currently selected folder. ; The following folder will be the root folder for the TreeView. Note that loading might take a long ; time if an entire drive such as C:\ is specified: TreeRoot := A_MyDocuments TreeViewWidth := 280 ListViewWidth := A_ScreenWidth/2 - TreeViewWidth - 30 ; Create the MyGui window and display the source directory (TreeRoot) in the title bar: MyGui := Gui("+Resize", TreeRoot) ; Allow the user to maximize or drag-resize the window. ; Create an ImageList and put some standard system icons into it: ImageListID := IL_Create(5) Loop 5 IL_Add(ImageListID, "shell32.dll", A_Index) ; Create a TreeView and a ListView side-by-side to behave like Windows Explorer: TV := MyGui.Add("TreeView", "r20 w" TreeViewWidth " ImageList" ImageListID) LV := MyGui.Add ("ListView", "r20 w" ListViewWidth " x+10", ["Name","Modified"]) ; Create a Status Bar to give info about the number of files and their total size: SB := MyGui.Add("StatusBar") SB.SetParts(60, 85) ; Create three parts in the bar (the third part fills all the remaining width). ; Add folders and their subfolders to the tree. Display the status in case loading takes a long time: M := Gui("ToolWindow -SysMenu Disabled AlwaysOnTop", "Loading the tree..."), M.Show("w200 h0") DirList := AddSubFoldersToTree(TreeRoot, Map()) M.Hide() ; Call TV_ItemSelect whenever a new item is selected: TV.OnEvent("ItemSelect", TV_ItemSelect) ; Call Gui_Size whenever the window is resized: MyGui.OnEvent("Size", Gui_Size) ; Set the ListView's column widths (this is optional): Col2Width := 70 ; Narrow to reveal only the YYYYMMDD part. LV.ModifyCol(1, ListViewWidth - Col2Width - 30) ; Allows room for vertical scrollbar. LV.ModifyCol(2, Col2Width) ; Display the window. The OS will notify the script whenever the user performs an eligible action: MyGui.Show AddSubFoldersToTree(Folder, DirList, ParentItemID := 0) { ; This function adds to the TreeView all subfolders in the specified folder ; and saves their paths associated with an ID into an object for later use. ; It also calls itself recursively to gather nested folders to any depth. Loop Files, Folder "*.*", "D" ; Retrieve all of Folder's sub-folders. { ItemID := TV.Add (A_LoopFileName, ParentItemID, "Icon4") DirList[ItemID] := A_LoopFilePath DirList := AddSubFoldersToTree(A_LoopFilePath, DirList, ItemID) } return DirList } TV_ItemSelect(thisCtrl, Item) ; This function is called when a new item is selected. { ; Put the files into the ListView: LV.Delete ; Clear all rows. LV.Opt ("-Redraw") ; Improve performance by disabling redrawing during load. TotalSize := 0 ; Init prior to loop below. Loop Files, DirList[Item] "*.*" ; For simplicity, omit folders so that only files are shown in the ListView. { LV.Add(, A_LoopFileName, A_LoopFileTimeModified) TotalSize += A_LoopFileSize } LV.Opt ("+Redraw") ; Update the three parts of the status bar to show info about the currently selected folder: SB.SetText(LV.GetCount() " files", 1) SB.SetText(Round (TotalSize / 1024, 1) " KB", 2) SB.SetText(DirList[Item], 3) } Gui_Size(thisGui, MinMax, Width, Height) ; Expand/Shrink ListView and TreeView in response to the user's resizing. { if MinMax = -1 ; The window has been minimized. No action needed. return ; Otherwise, the window has been resized or maximized. Resize the controls to match. TV.GetPos(,, &TV_W) TV.Move(,, Height - 30) ; -30 for StatusBar and margins. LV.Move(,, Width - TV_W - 30, Height - 30) } Trim / LTrim / RTrim - Syntax & Usage | AutoHotkey v2 Trim Trims characters from the beginning and/or end of a string. Result := Trim(String , OmitChars) Result := LTrim (String , OmitChars) Result := RTrim(String , OmitChars) Parameters String Type: String Any string value or variable. Numbers are not supported. OmitChars Type: String An optional list of characters (case sensitive) to exclude from the beginning and/or end of String. If omitted, spaces and tabs will be removed. Return Value Type: String These functions return the trimmed version of the specified string. Examples Trims all spaces from the left and right side of a string. text := " text " MsgBox ("No trim:`t'" text "' Trim:`t'" Trim(text) "' LTrim:`t'" LTrim(text) "' RTrim:`t'" RTrim(text) "'") Trims all zeros from the left side of a string. MsgBox LTrim("00000123", "0") Try - Syntax & Usage | AutoHotkey v2 Try Guards one or more statements against runtime errors and values thrown by the throw statement. Try Statement Try { Statements } Remarks The try statement is usually followed by a block - one or more statements (typically functions or expressions) enclosed in braces. If only a single statement is to be executed, it can be placed on the same line as try or on the next line, and the braces can be omitted. To specify code that executes only when try catches an error, use one or more catch statements. If try is used without catch or finally, it is equivalent to having catch Error with an empty block. A value can be thrown by the throw statement or by the program when a runtime error occurs. When a value is thrown from within a try block or a function called by one, the following occurs: If there is a catch statement which matches the class of the thrown value, execution is transferred into it. If there is no matching catch statement but there is a finally statement, it is executed, but once it finishes the value is automatically thrown again. If there is no matching catch statement or finally statement, the value is automatically thrown again (unless there is no catch or finally at all, as noted above). The last catch can optionally be followed by else. If the try statement completes without an exception being thrown (and without control being transferred elsewhere by return, break or similar), the else statement is executed. Exceptions thrown by the else statement are not handled by its associated try statement, but may be handled by an enclosing try statement. Finally can also be present, but must be placed after else, and is always executed last. The One True Brace (OTB) style may optionally be used with the try statement. For example: try { ... } catch Error as err { ... } else { ... } finally { ... } Related Throw, Catch, Else, Finally, Blocks, OnError Examples Demonstrates the basic concept of try/catch/throw. try ; Attempts to execute code. { HelloWorld MakeToast } catch as e ; Handles the first error thrown by the block above. { MsgBox "An error was thrown!`nSpecifically: " e.Message Exit } HelloWorld() ; Always succeeds. { MsgBox "Hello, world!" } MakeToast() ; Always fails. { ; Jump immediately to the try block's error handler: throw Error(A_ThisFunc " is not implemented, sorry") } Demonstrates basic error handling of built-in functions. try { ; The following tries to back up certain types of files: FileCopy A_MyDocuments "*.txt", "D:\Backup\Text documents" FileCopy A_MyDocuments "*.doc", "D:\Backup\Text documents" FileCopy A_MyDocuments "*.jpg", "D:\Backup\Photos" } catch { MsgBox "There was a problem while backing the files

up!"„ "IconX" ExitApp 1 } else { MsgBox "Backup successful." ExitApp 0 } Demonstrates the use of try/catch dealing with COM errors. For details about the COM object used below, see Using the ScriptControl (Microsoft Docs). try { obj := ComObject("ScriptControl") obj.ExecuteStatement('MsgBox "This is embedded VBScript"') ; This line produces an Error. obj.InvalidMethod() ; This line would produce a MethodError. } catch MemberError ; Covers MethodError and PropertyError. { MsgBox "We tried to invoke a member that doesn't exist." } catch as e { ; For more detail about the object that e contains, see Error Object. MsgBox("Exception thrown!`n`nwhat: " e.what "`nfile: " e.file . "`nline: " e.line "`nmessage: " e.message "`nextra: " e.extra„ 16) } Demonstrates nesting try-catch statements. try Example1 ; Any single statement can be on the same line with a Try statement. catch Number as e MsgBox "Example1() threw " e Example1() { try Example2 catch Number as e { if (e = 1) throw ; Rethrow the caught value to our caller. else MsgBox "Example2() threw " e } } Example2() { throw Random(1, 2) } Type - Syntax & Usage | AutoHotkey v2 Type Returns the class name of a value. ClassName := Type(Value) Return Value Type: String This function returns the class name of Value. The algorithm for determining a value's class name can be approximated as shown below: TypeOf(Value) { if (comClass := ComObjType (Value, "Class")) != "" return comClass try ; `Value is Object` is not checked because it can be false for prototypes. if ObjHasOwnProp(Value, "__Class") return "Prototype" while Value := ObjGetBase(Value) if ObjHasOwnProp(Value, "__Class") return Value.__Class return "Object" } For COM wrapper objects, the class name can also be determined based on the variant type, as follows: ComObject_Type(obj) { if ComObjType(obj) & 0x2000 ; VT_ARRAY return "ComObjArray" ; ComObjArray.Prototype.__Class if ComObjType(obj) & 0x4000 ; VT_BYREF return "ComValueRef" ; ComValueRef.Prototype.__Class if (ComObjType(obj) = 9 || ComObjType(obj) = 13) ; VT_DISPATCH || VT_UNKNOWN && ComObjValue(obj) != 0 { if (comClass := ComObjType(obj, "Class")) ! = "" return comClass if ComObjType(obj) = 9 ; VT_DISPATCH return "ComObject" ; ComObject.Prototype.__Class } return "ComValue" ; ComValue.Prototype.__Class } Remarks This function typically shouldn't be used to determine if a value is numeric, since numeric strings are valid in math expressions and with most built-in functions. However, in some cases the exact type of a value is more important. In such cases, consider using Value is Number or similar instead of Type. To check if a value can be used as a number, use the IsNumber, IsInteger or IsFloat function. To check for any type of object (that is, anything which is not a primitive value), use the IsObject function. To check if a value is an instance of a specific class, use the is operator. This can be used even with primitive values or to identify COM wrapper objects. Related Values, Expressions, Is functions, Integer, Float, String Examples Retrieves and reports the exact type of the values stored in a, b and c. a := 1, b := 2.0, c := "3" MsgBox Type(a) ; Integer MsgBox Type(b) ; Float MsgBox Type(c) ; String Until - Syntax & Usage | AutoHotkey v2 Until Applies a condition to the continuation of a Loop or For-loop. Loop { ... } Until Expression Parameters Expression Any valid expression. Remarks The space or tab after Until is optional if the expression is enclosed in parentheses, as in until(expression). The expression is evaluated once after each iteration, and is evaluated even if continue was used. If the expression evaluates to false (which is an empty string or the number 0), the loop continues; otherwise, the loop is broken and execution continues at the line following Until. Loop Until is shorthand for the following: Loop { ... if (Expression) break } However, Loop Until is often easier to understand and unlike the above, can be used with a single-line action. For example: Loop x *= 2 Until x > y Until can be used with any Loop or For. For example: Loop Read, A_ScriptFullPath lines .= A_LoopReadLine . "`n" Until A_Index=5 ; Read the first five lines. MsgBox lines If A_Index is used in Expression, it contains the index of the iteration which has just finished. Related Loop, While-loop, For-loop, Break, Continue, Blocks, Files-and-folders loop, Registry loop, File-reading loop, Parsing loop, If VarSetStrCapacity - Syntax & Usage | AutoHotkey v2 VarSetStrCapacity Enlarges a variable's holding capacity or frees its memory. This is not normally needed, but may be used with DllCall or SendMessage or to optimize repeated concatenation. GrantedCapacity := VarSetStrCapacity(&TargetVar , RequestedCapacity) Parameters &TargetVar Type: VarRef A reference to a variable. For example: VarSetStrCapacity(&MyVar, 1000). This can also be a dynamic variable such as Array%i% or a function's ByRef parameter. RequestedCapacity Type: Integer If omitted, the variable's current capacity will be returned and its contents will not be altered. Otherwise, anything currently in the variable is lost (the variable becomes blank). Specify for RequestedCapacity the number of characters that the variable should be able to hold after the adjustment. RequestedCapacity does not include the internal zero terminator. For example, specifying 1 would allow the variable to hold up to one character in addition to its internal terminator. Note: the variable will auto-expand if the script assigns it a larger value later. Since this function is often called simply to ensure the variable has a certain minimum capacity, for performance reasons, it shrinks the variable only when RequestedCapacity is 0. In other words, if the variable's capacity is already greater than RequestedCapacity, it will not be reduced (but the variable will still made blank for consistency). Therefore, to explicitly shrink a variable, first free its memory with VarSetStrCapacity(&Var, 0) and then use VarSetStrCapacity(&Var, NewCapacity) -- or simply let it auto-expand from zero as needed. For performance reasons, freeing a variable whose previous capacity was less than 64 characters might have no effect because its memory is of a permanent type. In this case, the current capacity will be returned rather than 0. For performance reasons, the memory of a variable whose capacity is less than 4096 bytes is not freed by storing an empty string in it (e.g. Var := ""). However, VarSetStrCapacity(&Var, 0) does free it. Specify -1 for RequestedCapacity to update the variable's internally-stored string length to the length of its current contents. This is useful in cases where the string has been altered indirectly, such as by passing its address via DllCall or SendMessage. In this mode, VarSetStrCapacity returns the length rather than the capacity. Return Value Type: Integer This function returns the number of characters that Var can now hold, which will be greater than or equal to RequestedCapacity. Failure An exception is thrown under any of the following conditions: TargetVar is not a valid variable reference. It is not possible to pass an object property or built-in variable by reference, and therefore not valid to pass one to this function. The requested capacity is too large to fit within any single contiguous memory block available to the script. In rare cases, this may be due to the system running out of memory. RequestedCapacity is less than -1 or greater than can theoretically be supported by the current platform. Remarks The Buffer object offers superior clarity and flexibility when dealing with binary data, structures, DllCall and similar. For instance, a Buffer object can be assigned to a property or array element or be passed to or returned from a function without copying its contents. This function can be used to enhance performance when building a string by means of gradual concatenation. This is because multiple automatic resizings can be avoided when you have some idea of what the string's final length will be. In such a case, RequestedCapacity need not be accurate: if the capacity is too small, performance is still improved and the variable will begin auto-expanding when the capacity has been exhausted. If the capacity is too large, some of the memory is wasted, but only temporarily because all the memory can be freed after the operation by means of VarSetStrCapacity(&Var, 0) or Var := "". Related Buffer object, DllCall, NumPut, NumGet Examples Optimize by ensuring MyVar has plenty of space to work with. VarSetStrCapacity(&MyVar, 5120000) ; ~10 MB Loop { ; ... MyVar .= StringToConcatenate ; ... } Use a variable to receive a string from an external function via DllCall. (Note that the use of a Buffer object may be preferred; in particular, when dealing with non-Unicode strings.) max_chars := 10 Loop 2 { ; Allocate space for use with DllCall. VarSetStrCapacity(&buf, max_chars) if (A_Index = 1) ; Alter the variable indirectly via DllCall. DllCall ("wsprintf", "Ptr", StrPtr(buf), "Str", "0x%08x", "UInt", 4919, "CDecl") else ; Use "str" to update the length automatically: DllCall("wsprintf", "Str", buf, "Str", "0x%08x", "UInt", 4919, "CDecl") ; Concatenate a string to demonstrate why the length needs to be updated: wrong_str := buf . "" wrong_len := StrLen (buf) ; Update the variable's length. VarSetStrCapacity(&buf, -1) right_str := buf . "" right_len := StrLen(buf) MsgBox ("Before updating String: " wrong_str " Length: " wrong_len " After updating String: " right_str " Length: " right_len) } VerCompare - Syntax & Usage | AutoHotkey v2 VerCompare Compares two version strings. Result := VerCompare(VersionA, VersionB) Parameters VersionA Type: String The first version string to be compared. VersionB Type: String The second version string to be compared, optionally prefixed with one of the following operators: <, <=, >, >= or =. Return Value Type: Integer If VersionB begins with an operator symbol, this function returns 1 (true) or 0 (false). Otherwise, this function returns one of the following to indicate the relationship between VersionA and VersionB: 0 if VersionA is equal to VersionB a positive integer if VersionA is greater than VersionB a negative integer if VersionA is less than VersionB To check for a specific relationship between the two strings, compare the result to 0. For example: a_less_than_b := VerCompare(a, b) < 0 a_greater_than_or_equal_to_b := VerCompare(a, b) >= 0 Remarks Version strings are compared according to the same rules as #Requires. This function should correctly compare Semantic Versioning 2.0.0 version strings, but the parameters are not required to be valid SemVer strings. This function can be used in a sort callback. Related #Requires, Sort, StrCompare Examples Checks the version of AutoHotkey in use. if VerCompare(A_AhkVersion, "2.0") < 0 MsgBox "This version < 2.0; possibly a pre-release version." else MsgBox "This version is 2.0 or later." Shows one difference between VerCompare and StrCompare. MsgBox VerCompare("1.20.0", "1.3") ; Returns 1 MsgBox StrCompare("1.20.0", "1.3") ; Returns -1 Demonstrates comparison with pre-release versions. MsgBox VerCompare("2.0-a137", "2.0-a136") ; Returns 1 MsgBox VerCompare("2.0-a137", "2.0") ; Returns -1 MsgBox VerCompare("10.2-beta.3", "10.2.0") ; Returns -1 Demonstrates a range check. MsgBox VerCompare("2.0.1", ">=2.0") && VerCompare("2.0.1", "<2.1") ; Returns 1 While Loop - Syntax & Usage | AutoHotkey v2 While-loop Performs a series of functions repeatedly until the specified expression evaluates to false. While Expression Parameters Expression Any valid expression. For example: while x < y. Remarks The space or tab after While is optional if the expression is enclosed in parentheses, as in While(expression). The expression is evaluated once before each iteration. If the expression evaluates to true (which is any result other than an empty string or the number 0), the body of the loop is executed; otherwise, execution jumps to the line following the loop's body. A while-loop is usually followed by a block, which is a collection of statements that form the body of the loop. However, a loop with only a single statement does not require a block (an "if" and its "else" count as a single statement for this purpose). The One True Brace (OTB) style may optionally be used, which allows the open-brace to appear on the same line rather than underneath. For example: while x < y {. The built-in variable A_Index contains the number of the current loop iteration. It contains 1 the first time the loop's expression and body are 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. As with all loops, Break may be used to exit the loop prematurely. Also, Continue may be used to skip the rest of the current iteration, at which time A_Index is increased by 1 and the while-loop's expression is re-evaluated. If it is still true, a new iteration begins; otherwise, the loop ends. The loop may optionally be followed by an Else statement, which is executed if the loop had zero iterations. Specialized loops: Loops can be used to automatically retrieve files, folders, or registry items (one at a time). See file-loop and registry-loop for details. In addition, file-reading loops can operate on the entire contents of a file,

one line at a time. Finally, parsing loops can operate on the individual fields contained inside a delimited string. Related Until, Break, Continue, Blocks, Loop, For-loop, Files-and-folders loop, Registry loop, File-reading loop, Parsing loop, If Examples As the user drags the left mouse button, a tooltip displays the size of the region inside the drag-area. CoordMode "Mouse", "Screen" ~LButton:: { MouseGetPos &begin_x, &begin_y while GetKeyState("LButton") { MouseGetPos &x, &y ToolTip begin_x ", " begin_y "`n" Abs(begin_x-x) " x " Abs(begin_y-y) Sleep 10 } ToolTip } List of Win Functions | AutoHotkey v2 Win Functions Functions to retrieve information about one or more windows, or make a variety of changes to a window. Click on a function name for details. Function Description WinActivate Activates the specified window. WinActivateBottom Same as WinActivate except that it activates the bottommost matching window rather than the topmost. WinActive Checks if the specified window exists and is currently active (foremost). WinClose Closes the specified window. WinExist Checks if the specified window exists. WinGetClass Retrieves the specified window's class name. WinGetClientPos Retrieves the position and size of the specified window's client area. WinGetControls Returns the control names for all controls in the specified window. WinGetControlsHwnd Returns the unique ID numbers for all controls in the specified window. WinGetCount Returns the number of existing windows that match the specified criteria. WinGetID Returns the unique ID number of the specified window. WinGetIDLast Returns the unique ID number of the last/bottommost window if there is more than one match. WinGetList Returns the unique ID numbers of all existing windows that match the specified criteria. WinGetMinMax Returns the state whether the specified window is maximized or minimized. WinGetPID Returns the Process ID number of the specified window. WinGetPos Retrieves the position and size of the specified window. WinGetProcessName Returns the name of the process that owns the specified window. WinGetProcessPath Returns the full path and name of the process that owns the specified window. WinGetStyleWinGetExStyle Returns the style or extended style (respectively) of the specified window. WinGetText Retrieves the text from the specified window. WinGetTitle Retrieves the title of the specified window. WinGetTransColor Returns the color that is marked transparent in the specified window. WinGetTransparent Returns the degree of transparency of the specified window. WinHide Hides the specified window. WinKill Forces the specified window to close. WinMaximize Enlarges the specified window to its maximum size. WinMinimize Collapses the specified window into a button on the task bar. WinMinimizeAllWinMinimizeAllUndo Minimizes or unminimizes all windows. WinMove Changes the position and/or size of the specified window. WinMoveBottom Sends the specified window to the bottom of stack; that is, beneath all other windows. WinMoveTop Brings the specified window to the top of the stack without explicitly activating it. WinRedraw Redraws the specified window. WinRestore Unminimizes or unmaximizes the specified window if it is minimized or maximized. WinSetAlwaysOnTop Makes the specified window stay on top of all other windows (except other always-on-top windows). WinSetEnabled Enables or disables the specified window. WinSetRegion Changes the shape of the specified window to be the specified rectangle, ellipse, or polygon. WinSetStyleWinSetExStyle Changes the style or extended style of the specified window, respectively. WinSetTitle Changes the title of the specified window. WinSetTransColor Makes all pixels of the chosen color invisible inside the specified window. WinSetTransparent Makes the specified window semi-transparent. WinShow Unhides the specified window. WinWait Waits until the specified window exists. WinWaitActiveWinWaitNotActive Waits until the specified window is active or not active. WinWaitClose Waits until no matching windows can be found. Remarks To discover the unique ID number of the window that the mouse is currently hovering over, use MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related SetWinDelay, Control functions, Gui object (for windows created by the script) WinActivate - Syntax & Usage | AutoHotkey v2 WinActivate Activates the specified window. WinActivate WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. Remarks When an inactive window becomes active, the operating system also makes it foremost (brings it to the top of the stack). This does not occur if the window is already active. If the window is minimized and inactive, it is automatically restored prior to being activated. If WinTitle is the letter "A" and the other parameters are omitted, the active window is restored. The window is restored even if it was already active. Six attempts will be made to activate the target window over the course of 60ms. If all six attempts fail, WinActivate automatically sends {Alt 2} as a workaround for possible restrictions enforced by the operating system, and then makes a seventh attempt. Thus, it is usually unnecessary to follow WinActivate with WinWaitActive or if not WinActive(...). After WinActivate's first failed attempt, it automatically sends {Alt up}. Testing has shown that this may improve the reliability of all subsequent attempts, reducing the number of instances where the first attempt fails and causes the taskbar button to flash. No more than one {Alt up} is sent for this purpose for the lifetime of each script. If this or any other (AutoHotkey v1.1.27+) script has a keyboard hook installed, the {Alt up} is blocked from the active window, minimizing the already small risk of side-effects. In general, if more than one window matches, the topmost matching window (typically the one most recently used) will be activated. If the window is already active, it will be kept active rather than activating any other matching window beneath it. However, if the active window is moved to the bottom of the stack with WinMoveBottom, some other window may be activated even if the active window is a match. WinActivateBottom activates the bottommost matching window (typically the one least recently used). GroupActivate activates the next window that matches criteria specified by a window group. If the active window is hidden and DetectHiddenWindows is turned off, it is never considered a match. Instead, a visible matching window is activated if one exists. When a window is activated immediately after the activation of some other window, task bar buttons might start flashing on some systems (depending on OS and settings). To prevent this, use #WinActivateForce. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Known issue: If the script is running on a computer or server being accessed via remote desktop, WinActivate may hang if the remote desktop client is minimized. One workaround is to use built-in functions which don't require window activation, such as ControlSend and ControlClick. Another possible workaround is to apply the following registry setting on the local/client computer: ; Change HKCU to HKLM to affect all users on this system. RegWrite "REG_DWORD", "HKCU\Software\Microsoft\Terminal Server Client" , "RemoteDesktop_SuppressWhenMinimized", 2 Related WinActivateBottom, #WinActivateForce, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, WinExist, WinActive, WinWaitActive, WinWait, WinWaitClose, WinClose, GroupActivate, Win functions Examples If Notepad does exist, activate it, otherwise activate the calculator. if WinExist ("Untitled - Notepad") WinActivate ; Use the window found by WinExist. else WinActivate "Calculator" WinActivateBottom - Syntax & Usage | AutoHotkey v2 WinActivateBottom Same as WinActivate except that it activates the bottommost matching window rather than the topmost. WinActivateBottom WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. Remarks The bottommost window is typically the one least recently used, except when windows have been reordered, such as with WinMoveBottom. If there is only one matching window, WinActivateBottom behaves identically to WinActivate. Window groups are more advanced than this function, so consider using them for more features and flexibility. If the window is minimized and inactive, it is automatically restored prior to being activated. If WinTitle is the letter "A" and the other parameters are omitted, the active window is restored. The window is restored even if it was already active. Six attempts will be made to activate the target window over the course of 60ms. Thus, it is usually unnecessary to follow it with the WinWaitActive function. Unlike WinActivate, the Last Found Window cannot be used because it might not be the bottommost window. Therefore, at least one of the parameters must be non-blank. When a window is activated immediately after another window was activated, task bar buttons may start flashing on some systems (depending on OS and settings). To prevent this, use #WinActivateForce. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinActivate, #WinActivateForce, SetTitleMatchMode, DetectHiddenWindows, WinExist, WinActive, WinWaitActive, WinWait, WinWaitClose, GroupActivate Examples Press a hotkey to visit all open browser windows in order from oldest to newest. #i:: ; Win+I { SetTitleMatchMode 2 WinActivateBottom "- Microsoft Internet Explorer" } WinActive - Syntax & Usage | AutoHotkey v2 WinActive Checks if the specified window is active and returns its unique ID (HWND). UniqueID := WinActive(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the unique ID (HWND) of the active window if it matches the specified criteria, or 0 if it does not. Since all non-zero numbers are seen as "true", the statement if WinActive(WinTitle) is true whenever WinTitle is active. Remarks If all parameters are omitted, the Last Found Window will be used. If the active window is a qualified match, the Last Found Window will be updated to be the active window. An easy way to retrieve the unique ID of the active window is with ActiveHwnd := WinExist("A"). SetWinDelay does not apply to this function. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinExist, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, WinActivate, WinWaitActive, WinWait, WinWaitClose, #HotIf Examples Closes either Notepad or another window, depending on which of them was found by the WinActive functions above. Note that the space between an "ahk_" keyword and its criterion value can be omitted; this is especially useful when using variables, as shown by the second WinActive. if WinActive("ahk_class Notepad") or WinActive("ahk_class" ClassName) WinClose ; Use the window found by WinActive. WinClose - Syntax & Usage | AutoHotkey v2 WinClose Closes the specified window. WinClose WinTitle, WinText, SecondsToWait, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must

be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. SecondsToWait Type: Integer or Float If omitted or blank, the function will not wait at all. If 0, it will wait 500ms. Otherwise, it will wait the indicated number of seconds (can contain a decimal point or be an expression) for the window to close. If the window does not close within that period, the script will continue. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. No exception is thrown if a window is found but cannot be closed, so use WinExist or WinWaitClose if you need to determine for certain that a window has closed. Remarks This function sends a close message to a window. The result depends on the window (it may ask to save data, etc.) If a matching window is active, that window will be closed in preference to any other matching window. In general, if more than one window matches, the topmost (most recently used) will be closed. This function operates only upon a single window except when WinTitle is ahk_group GroupName (with no other criteria specified), in which case all windows in the group are affected. WinClose sends a WM_CLOSE message to the target window, which is a somewhat forceful method of closing it. An alternate method of closing is to send the following message. It might produce different behavior because it is similar in effect to pressing Alt+F4 or clicking the window's close button in its title bar: PostMessage 0x0112, 0xF060,,, WinTitle, WinText ; 0x0112 = WM_SYSCOMMAND, 0xF060 = SC_CLOSE If a window does not close via WinClose, you can force it to close with WinKill. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinKill, WinWaitClose, ProcessClose, WinActivate, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, WinExist, WinActive, WinWaitActive, WinWait, GroupActivate Examples If Notepad does exist, close it, otherwise close the calculator. if WinExist("Untitled - Notepad") WinClose ; Use the window found by WinExist. else WinClose "Calculator" WinExist - Syntax & Usage | AutoHotkey v2 WinExist Checks if the specified window exists and returns the unique ID (HWND) of the first matching window. UniqueID := WinExist(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the unique ID (HWND) of the first matching window (or 0 if none). Since all non-zero numbers are seen as "true", the statement if WinExist (WinTitle) is true whenever WinTitle exists. Remarks If all parameters are omitted, the Last Found Window will be checked to see if it still exists. If a qualified window exists, the Last Found Window will be updated to be that window. To discover the HWND of a control (for use with PostMessage, SendMessage or DllCall), use ControlGetHwnd or MouseGetPos. SetWinDelay does not apply to this function. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinActive, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, ProcessExist, WinActivate, WinWaitActive, WinWait, WinWaitClose, #HotIf Examples Activates either Notepad or another window, depending on which of them was found by the WinExist functions above. Note that the space between an "ahk_" keyword and its criterion value can be omitted; this is especially useful when using variables, as shown by the second WinExist. if WinExist("ahk_class Notepad") or WinExist("ahk_class" ClassName) WinActivate ; Use the window found by WinExist. Retrieves and reports the unique ID (HWND) of the active window. MsgBox "The active window's ID is " WinExist("A") Returns if the calculator does not exist. if not WinExist("Calculator") return WinGetClass - Syntax & Usage | AutoHotkey v2 WinGetClass Retrieves the specified window's class name. Class := WinGetClass(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the class name of the specified window. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown on if the class name could not be retrieved. Remarks Only the class name is retrieved (the prefix "ahk_class" is not included in the return value). For a general explanation of window classes and one way to use them, see ahk_class. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetTitle, Win functions Examples Retrieves and reports the class name of the active window. MsgBox "The active window's class is " WinGetClass ("A") WinGetClientPos - Syntax & Usage | AutoHotkey v2 WinGetClientPos Retrieves the position and size of the specified window's client area. WinGetClientPos &X, &Y, &Width, &Height, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters &X, &Y Type: VarRef References to output variables in which to store the X and Y coordinates of the client area's upper left corner. If omitted, the corresponding values will not be stored. &Width, &Height Type: VarRef References to output variables in which to store the width and height of the client area. If omitted, the corresponding values will not be stored. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. Remarks The client area is the part of the window which can contain controls. It excludes the window's title bar, menu (if it has a standard one) and borders. The position and size of the client area are less dependent on OS version and theme than the values returned by WinGetPos. The values returned for a minimized window may vary depending on OS and configuration, but are usually -32000 for the X and Y coordinates and zero for the width and height. Related WinGetPos, WinMove, ControlGetPos, WinGetTitle, WinGetText, ControlGetText Examples Retrieves and reports the position and size of the calculator's client area. WinGetClientPos &X, &Y, &W, &H, "Calculator" MsgBox "Calculator's client area is at " X "," Y " and its size is " W "x" H Retrieves and reports the position of the active window's client area. WinGetClientPos &X, &Y,,, "A" MsgBox "The active window's client area is at " X "," Y If Notepad does exist, retrieve and report the position of its client area. if WinExist("Untitled - Notepad") { WinGetClientPos &Xpos, &Ypos ; Use the window found by WinExist. MsgBox "Notepad's client area is at " Xpos "," Ypos } WinGetControls - Syntax & Usage | AutoHotkey v2 WinGetControls Returns the control names for all controls in the specified window. Controls := WinGetControls(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Array of Strings This function returns an array containing the control names for all controls in the specified window. Each name of a control consists of its class name followed immediately by its sequence number (ClassNN), as shown by Window Spy. For example, if the return value is assigned to a variable named Controls and two controls are present, Controls[1] contains the name of the first control, Controls[2] contains the name of the second control, and Controls.Length returns the number 2. Controls are sorted according to their Z-order, which is usually the same order as the navigation order via Tab if the window supports tabbing. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window or control under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetControlsHwnd, Win functions, Control functions Examples Extracts the individual control names from the active window's control list. for n, ctrl in WinGetControls("A") { Result := MsgBox("Control #" n " is '" ctrl "'. Continue?",, 4) if (Result = "No") break } Displays in real time the active window's control list. SetTimer WatchActiveWindow, 200 WatchActiveWindow() { try { Controls := WinGetControls("A") ControlList := "" for ClassNN in Controls ControlList .= ClassNN . "`n" if (ControlList = "") ToolTip "The active window has no controls." else ToolTip ControlList } catch TargetError ToolTip "No visible window is active." } WinGetControlsHwnd - Syntax & Usage | AutoHotkey v2 WinGetControlsHwnd Returns the unique ID numbers for all controls in the specified window. Controls := WinGetControlsHwnd(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Array of Integers This function returns an array containing the window handles (HWND) for all controls in the specified window. For example, if the return value is assigned to a variable named Controls and two controls are present, Controls[1] contains the ID of the first control, Controls[2] contains the ID of the second control, and Controls.Length returns the number 2. Controls are sorted according to their Z-order, which is usually the same order as the navigation order via Tab if the window supports tabbing. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window or control under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetControls, Win functions, Control functions WinGetCount - Syntax & Usage | AutoHotkey v2 WinGetCount Returns the number of existing windows that match the specified criteria. Count := WinGetCount(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the number of existing windows that match the specified parameters. If there is no matching window, zero is returned. Remarks To count all

windows on the system, omit all four title/text parameters. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetList, Win functions, Control functions WinGetID - Syntax & Usage | AutoHotkey v2 WinGetID Returns the unique ID number of the specified window. ID := WinGetID(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the window handle (HWND) of the specified window. Error Handling A TargetError is thrown if the window could not be found. Remarks This function is equivalent to WinExist. A window's ID number is valid only during its lifetime. In other words, if an application restarts, all of its windows will get new ID numbers. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. To discover the HWND of a control (for use with PostMessage, SendMessage or DllCall), use ControlGetHwnd or MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetIDLast, ControlGetHwnd, Hwnd property (Gui object), Win functions, Control functions Examples Maximizes the active window and reports its unique ID. active_id := WinGetID("A") WinMaximize active_id MsgBox "The active window's ID is " active_id WinGetIDLast - Syntax & Usage | AutoHotkey v2 WinGetIDLast Returns the unique ID number of the last/bottommost window if there is more than one match. IDLast := WinGetIDLast(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the window handle (HWND) of the last/bottommost window if there is more than one match. Error Handling A TargetError is thrown if the window could not be found. Remarks If there is only one match, it performs identically to WinGetID. A window's ID number is valid only during its lifetime. In other words, if an application restarts, all of its windows will get new ID numbers. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. To discover the HWND of a control (for use with PostMessage, SendMessage or DllCall), use ControlGetHwnd or MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetID, ControlGetHwnd, Hwnd property (Gui object), Win functions, Control functions WinGetList - Syntax & Usage | AutoHotkey v2 WinGetList Returns the unique ID numbers of all existing windows that match the specified criteria. List := WinGetList(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Array of Integers This function returns an array containing the window handles (HWND) of all existing windows that match the specified parameters. If there is no matching window, an empty array is returned. For example, if the return value is assigned to a variable named List and two matching windows are discovered, List[1] contains the ID of the first window, List[2] contains the ID of the second window, and List.Length returns the number 2. Windows are retrieved in order from topmost to bottommost (according to how they are stacked on the desktop). Remarks To retrieve all windows on the entire system, omit all four title/text parameters. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetCount, Win functions, Control functions Examples Visits all windows on the entire system and displays info about each of them. ids := WinGetList(,, "Program Manager") for this_id in ids { WinActivate this_id this_class := WinGetClass(this_id) this_title := WinGetTitle(this_id) Result := MsgBox(("Visiting All Windows " A_Index " of " ids.Length " ahk_id " this_id " ahk_class " this_class " " this_title " Continue?"),, 4) if (Result = "No") break } WinGetMinMax - Syntax & Usage | AutoHotkey v2 WinGetMinMax Returns the state whether the specified window is maximized or minimized. MinMax := WinGetMinMax(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the current state of the specified window: -1: The window is minimized (WinRestore can unminimize it). 1: The window is maximized (WinRestore can unmaximize it). 0: The window is neither minimized nor maximized. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMaximize, WinMinimize, Win functions, Control functions WinGetPID - Syntax & Usage | AutoHotkey v2 WinGetPID Returns the Process ID number of the specified window. PID := WinGetPID(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the Process ID (PID) of the specified window. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetProcessName, WinGetProcessPath, Process functions, Win functions, Control functions WinGetPos - Syntax & Usage | AutoHotkey v2 WinGetPos Retrieves the position and size of the specified window. WinGetPos &OutX, &OutY, &OutWidth, &OutHeight, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters &OutX, &OutY Type: VarRef References to the output variables in which to store the X and Y coordinates of the target window's upper left corner. If omitted, the corresponding values will not be stored. &OutWidth, &OutHeight Type: VarRef References to the output variables in which to store the width and height of the target window. If omitted, the corresponding values will not be stored. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. Remarks If the WinTitle "Program Manager" is used, the function will retrieve the size of the desktop, which is usually the same as the current screen resolution. A minimized window will still have a position and size. The values returned in this case may vary depending on OS and configuration. To discover the name of the window and control that the mouse is currently hovering over, use MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. As the coordinates returned by this function include the window's title bar, menu and borders, they may be dependent on OS version and theme. To get more consistent values across different systems, consider using WinGetClientPos instead. On systems with multiple screens which have different DPI settings, the returned position and size may be different than expected due to OS DPI scaling. Related WinMove, WinGetClientPos, ControlGetPos, WinGetTitle, WinGetText, ControlGetText Examples Retrieves and reports the position and size of the calculator. WinGetPos &X, &Y, &W, &H, "Calculator" MsgBox "Calculator is at " X "," Y " and its size is " W "x" H Retrieves and reports the position of the active window. WinGetPos &X, &Y,,, "A" MsgBox "The active window is at " X "," Y If Notepad does exist, retrieve and report its position. if WinExist("Untitled - Notepad") { WinGetPos &Xpos, &Ypos ; Use the window found by WinExist. MsgBox "Notepad is at " Xpos "," Ypos } WinGetProcessName - Syntax & Usage | AutoHotkey v2 WinGetProcessName Returns the name of the process that owns the specified window. ProcessName := WinGetProcessName (WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the process name of the specified window. For example: notepad.exe. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetPID, WinGetProcessPath, Process functions, Win functions, Control functions WinGetProcessPath - Syntax & Usage | AutoHotkey v2 WinGetProcessPath Returns the full path and name of the process that owns the specified window. ProcessPath := WinGetProcessPath(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the full path and name of the process that owns the specified window. For example: C:\Windows\System32\notepad.exe. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window

titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetPID, WinGetProcessName, Process functions, Win functions, Control functions WinGetStyle / WinGetExStyle - Syntax & Usage | AutoHotkey v2 WinGetStyle / WinGetExStyle Returns the style or extended style (respectively) of the specified window. Style := WinGetStyle(WinTitle, WinText, ExcludeTitle, ExcludeText) ExStyle := WinGetExStyle (WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer These functions return the style or extended style (respectively) of the specified window. If there is no matching window, an empty string is returned. Error Handling A TargetError is thrown if the window could not be found. Remarks See the styles table for a partial listing of styles. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinSetStyle / WinSetExStyle, ControlGetStyle / ControlGetExStyle, styles table, Win functions, Control functions Examples Determines whether a window has the WS_DISABLED style. Style := WinGetStyle("My Window Title") if (Style & 0x8000000) ; 0x8000000 is WS_DISABLED. MsgBox "The window is disabled." Determines whether a window has the WS_EX_TOPMOST style (always-on-top). ExStyle := WinGetExStyle("My Window Title") if (ExStyle & 0x8) ; 0x8 is WS_EX_TOPMOST. MsgBox "The window is always-on-top." WinGetText - Syntax & Usage | AutoHotkey v2 WinGetText Retrieves the text from the specified window. Text := WinGetText(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the text of the specified window. Error Handling A TargetError is thrown if the window could not be found. An Error is thrown if there was a problem retrieving the window's text. Remarks The text retrieved is generally the same as what Window Spy shows for that window. However, if DetectHiddenText has been turned off, hidden text is omitted from the return value. Each text element ends with a carriage return and linefeed (CR+LF), which can be represented in the script as `r`n. To extract individual lines or substrings, use functions such as InStr and SubStr. A parsing loop can also be used to examine each line or word one by one. If the retrieved text appears to be truncated (incomplete), it may be necessary to retrieve the text by sending the WM_GETTEXT message via SendMessage instead. This is because some applications do not respond properly to the WM_GETTEXTLENGTH message, which causes AutoHotkey to make the return value too small to fit all the text. This function might use a large amount of RAM if the target window (e.g. an editor with a large document open) contains a large quantity of text. To avoid this, it might be possible to retrieve only portions of the window's text by using ControlGetText instead. In any case, a variable's memory can be freed later by assigning it to nothing, i.e. Text := "". To retrieve a list of all controls in a window, follow this example: Controls := WinGetControls(WinTitle) Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlGetText, WinGetTitle, WinGetPos Examples Opens the calculator, waits until it exists, and retrieves and reports its text. Run "calc.exe" WinWait "Calculator" MsgBox "The text is:`n" WinGetText() ; Use the window found by WinWait. WinGetTitle - Syntax & Usage | AutoHotkey v2 WinGetTitle Retrieves the title of the specified window. Title := WinGetTitle(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the title of the specified window. If there is no matching window, an empty string is returned. Error Handling A TargetError is thrown if the window could not be found. Remarks To discover the name of the window that the mouse is currently hovering over, use MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinSetTitle, WinGetClass, WinGetText, ControlGetText, WinGetPos, Win functions Examples Retrieves and reports the title of the active window. MsgBox "The active window is '" WinGetTitle("A") "'." WinGetTransColor - Syntax & Usage | AutoHotkey v2 WinGetTransColor Returns the color that is marked transparent in the specified window. TransColor := WinGetTransColor(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: String This function returns the six-digit RGB color such as 0x00CC99 that is marked transparent in the specified window (see WinSetTransColor for how to set the TransColor). The return value is an empty string if: 1) there are no matching windows; 2) the window has no transparency color; or 3) other conditions (caused by OS behavior) such as the window having been minimized, restored, and/or resized since it was made transparent. Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinSetTransColor, WinGetTransparent, Win functions, Control functions Examples Retrieves the transparent color of a window under the mouse cursor. MouseGetPos ,, &MouseWin TransColor := WinGetTransColor (MouseWin) WinGetTransparent - Syntax & Usage | AutoHotkey v2 WinGetTransparent Returns the degree of transparency of the specified window. Transparent := WinGetTransparent(WinTitle, WinText, ExcludeTitle, ExcludeText) Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer or String (empty) The return value is an empty string if the window has no transparency level, or under other conditions (caused by OS behavior) such as the window having been minimized, restored, and/or resized since it was made transparent. Otherwise, the return value is an integer between 0 and 255 representing the degree of transparency of the specified window, where 0 indicates an invisible window and 255 indicates an opaque window (see WinSetTransparent for how to set transparency). Error Handling A TargetError is thrown if the window could not be found. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinSetTransparent, WinGetTransColor, Win functions, Control functions Examples Retrieves the transparency of a window under the mouse cursor. MouseGetPos ,, &MouseWin Transparent := WinGetTransparent(MouseWin) WinHide - Syntax & Usage | AutoHotkey v2 WinHide Hides the specified window. WinHide WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. Remarks Use WinShow to unhide a hidden window (DetectHiddenWindows can be either On or Off to do this). This function operates only upon the topmost matching window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. The Explorer taskbar may be hidden/shown as follows: WinHide "ahk_class Shell_TrayWnd" WinShow "ahk_class Shell_TrayWnd" Related WinShow, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, Win functions Examples Opens Notepad, waits until it exists, hides it for a short time and unhides it. Run "notepad.exe" WinWait "Untitled - Notepad" Sleep 500 WinHide ; Use the window found by WinWait. Sleep 1000 WinShow ; Use the window found by WinWait. WinKill - Syntax & Usage | AutoHotkey v2 WinKill Forces the specified window to close. WinKill WinTitle, WinText, SecondsToWait, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. SecondsToWait Type: Integer or Float If omitted or blank, the function will not wait at all. If 0, it will wait 500ms. Otherwise, it will wait the indicated number of seconds (can contain a decimal point or be an expression) for the window to close. If the window does not close within that period, the script will continue. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. No exception is thrown if a window is found but cannot be closed, so use WinExist or WinWaitClose if you need to determine for certain that a window has closed. Remarks This function first makes a brief attempt to close the window normally. If that fails, it will attempt to force the window closed by terminating its process. If a matching window is active, that window will be closed in preference to any other matching window. In general, if more than one window matches, the topmost (most recently used) will be closed. This function operates only upon a single window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinClose, WinWaitClose, ProcessClose, WinActivate, SetTitleMatchMode, DetectHiddenWindows, Last Found Window, WinExist, WinActive, WinWaitActive, WinWait, GroupActivate Examples If Notepad does exist, force it to close, otherwise force the calculator to close. if WinExist("Untitled - Notepad") WinKill ; Use the window found by WinExist. else WinKill "Calculator" WinMaximize - Syntax & Usage | AutoHotkey v2

WinMaximize Enlarges the specified window to its maximum size. WinMaximize WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. Remarks Use WinRestore to unmaximize a window and WinMinimize to minimize it. If a particular type of window does not respond correctly to WinMaximize, try using the following instead: PostMessage 0x0112, 0xF030,,, WinTitle, WinText ; 0x0112 = WM_SYSCOMMAND, 0xF030 = SC_MAXIMIZE This function operates only upon the topmost matching window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinRestore, WinMinimize Examples Opens Notepad, waits until it exists and maximizes it. Run "notepad.exe" WinWait "Untitled - Notepad" WinMaximize ; Use the window found by WinWait. Press a hotkey to maximize the active window. ^Up::WinMaximize "A" ; Ctrl+Up WinMinimize - Syntax & Usage | AutoHotkey v2 WinMinimize Collapses the specified window into a button on the task bar. WinMinimize WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. Remarks Use WinRestore or WinMaximize to unminimize a window. WinMinimize minimizes the window using a direct method, bypassing the window message which is usually sent when the minimize button, window menu or taskbar is used to minimize the window. This prevents the window from overriding the action (such as to "minimize" to the taskbar by hiding the window), but may also prevent the window from responding correctly, such as to save the current focus for when the window is restored. It also prevents the "minimize" system sound from being played. If a particular type of window does not respond correctly to WinMinimize, try using the following instead: PostMessage 0x0112, 0xF020,,, WinTitle, WinText ; 0x0112 = WM_SYSCOMMAND, 0xF020 = SC_MINIMIZE This function operates only upon the topmost matching window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinRestore, WinMaximize, WinMinimizeAll Examples Opens Notepad, waits until it exists and minimizes it. Run "notepad.exe" WinWait "Untitled - Notepad" WinMinimize ; Use the window found by WinWait. Press a hotkey to minimize the active window. ^Down::WinMinimize "A" ; Ctrl+Down WinMinimizeAll / WinMinimizeAllUndo - Syntax & Usage | AutoHotkey v2 WinMinimizeAll / WinMinimizeAllUndo Minimizes or unminimizes all windows. WinMinimizeAll WinMinimizeAllUndo On most systems, this is equivalent to Explorer's Win+M and Win+D hotkeys. Related WinMinimize, GroupAdd Examples Minimizes all windows for 1 second and unminimizes them. WinMinimizeAll Sleep 1000 WinMinimizeAllUndo WinMove - Syntax & Usage | AutoHotkey v2 WinMove Changes the position and/or size of the specified window. WinMove X, Y, Width, Height, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters X, Y Type: Integer The X and Y coordinates (in pixels) of the upper left corner of the target window's new location. The upper-left pixel of the screen is at 0, 0. If these are the only parameters given with the function, the Last Found Window will be used as the target window. Otherwise, X and/or Y can be omitted, in which case the current position is used. Width, Height Type: Integer The new width and height of the window (in pixels). If either is omitted or blank, the size in that dimension will not be changed. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if an internal function call reported failure. However, success may be reported even if the window has not moved, such as if the window restricts its own movement. Remarks If Width and Height are small (or negative), most windows with a title bar will generally go no smaller than 112 x 27 pixels (however, some types of windows may have a different minimum size). If Width and Height are large, most windows will go no larger than approximately 12 pixels beyond the dimensions of the desktop. Negative values are allowed for the x and y coordinates to support multi-monitor systems and to allow a window to be moved entirely off screen. Although WinMove cannot move minimized windows, it can move hidden windows if DetectHiddenWindows is on. The speed of WinMove is affected by SetWinDelay. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. On systems with multiple screens which have different DPI settings, the final position and size of the window may differ from the requested values due to OS DPI scaling. Related ControlMove, WinGetPos, WinHide, WinMinimize, WinMaximize, Win functions Examples Opens the calculator, waits until it exists and moves it to the upper-left corner of the screen. Run "calc.exe" WinWait "Calculator" WinMove 0, 0 ; Use the window found by WinWait. Creates a fixed-size popup window that shows the contents of the clipboard and moves it to the upper-left corner of the screen. MyGui := Gui("ToolWindow -Sysmenu Disabled", "The clipboard contains:") MyGui.Add("Text",, A_Clipboard) MyGui.Show("w400 h300") WinMove 0, 0,,, MyGui.Title ; Move the splash window to the top left corner. MsgBox "Press OK to dismiss the MyGui window" MyGui.Destroy Centers a window on the screen. CenterWindow("ahk_class Notepad") CenterWindow(WinTitle) { WinGetPos ,, &Width, &Height, WinTitle WinMove (A_ScreenWidth/2)-(Width/2), (A_ScreenHeight/2)-(Height/2),,, WinTitle } WinMoveBottom - Syntax & Usage | AutoHotkey v2 WinMoveBottom Sends the specified window to the bottom of stack; that is, beneath all other windows. WinMoveBottom WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError may be thrown on failure. Remarks The effect is similar to pressing Alt+Esc. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMoveTop, WinSetAlwaysOnTop, Win functions, Control functions WinMoveTop - Syntax & Usage | AutoHotkey v2 WinMoveTop Brings the specified window to the top of the stack without explicitly activating it. WinMoveTop WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError may be thrown on failure. Remarks However, the system default settings will probably cause it to activate in most cases. In addition, this function may have no effect due to the operating system's protection against applications that try to steal focus from the user (it may depend on factors such as what type of window is currently active and what the user is currently doing). One possible work-around is to make the window briefly always-on-top via WinSetAlwaysOnTop, then turn off always-on-top. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMoveBottom, WinSetAlwaysOnTop, Win functions, Control functions WinRedraw - Syntax & Usage | AutoHotkey v2 WinRedraw Redraws the specified window. WinRedraw WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. Remarks This function attempts to update the appearance/contents of a window by informing the OS that the window's rectangle needs to be redrawn. If this approach does not work for a particular window, try WinMove. If that does not work, try WinHide in combination with WinShow. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMoveBottom, WinSetAlwaysOnTop, Win functions, Control functions WinRestore - Syntax & Usage | AutoHotkey v2 WinRestore Unminimizes or unmaximizes the specified window if it is minimized or maximized. WinRestore WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. Remarks If a particular type of window does not respond correctly to WinRestore, try using the following instead: PostMessage 0x0112, 0xF120,,, WinTitle, WinText ; 0x0112 = WM_SYSCOMMAND, 0xF120 = SC_RESTORE This function operates only upon the topmost matching window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMinimize, WinMaximize Examples Unminimizes or unmaximizes Notepad if it is minimized or maximized. WinRestore "Untitled - Notepad" WinSetAlwaysOnTop - Syntax & Usage | AutoHotkey v2 WinSetAlwaysOnTop Makes the specified window stay on top of all other windows (except other always-on-top windows). WinSetAlwaysOnTop Value, WinTitle, WinText, ExcludeTitle,

ExcludeText Parameters Value Type: Integer One of the following values: 1 or True turns on the setting 0 or False turns off the setting -1 sets it to the opposite of its current state WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError may be thrown on failure. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMoveTop, WinMoveBottom, Win functions, Control functions Examples Toggles the always-on-top status of the calculator. WinSetAlwaysOnTop -1, "Calculator" WinSetEnabled - Syntax & Usage | AutoHotkey v2 WinSetEnabled Enables or disables the specified window. WinSetEnabled Value , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Value Type: Integer One of the following values: 1 or True turns on the setting 0 or False turns off the setting -1 sets it to the opposite of its current state WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if the change could not be applied. Remarks When a window is disabled, the user cannot move it or interact with its controls. In addition, disabled windows are omitted from the alt-tab list. Example #1 on the WinGetStyle page can be used to determine whether a window is disabled. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related ControlSetEnabled, Win functions, Control functions WinSetRegion - Syntax & Usage | AutoHotkey v2 WinSetRegion Changes the shape of the specified window to be the specified rectangle, ellipse, or polygon. WinSetRegion Options, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Options Type: String If blank or omitted, the window is restored to its original/default display area. Otherwise, one or more of the following options can be specified, each separated from the others with space(s): Wn: Width of rectangle or ellipse. For example: w200. Hn: Height of rectangle or ellipse. For example: h200. X-Y: Each of these is a pair of X/Y coordinates. For example, 200-0 would use 200 for the X coordinate and 0 for the Y. E: Makes the region an ellipse rather than a rectangle. This option is valid only when W and H are present. R [w-h]: Makes the region a rectangle with rounded corners. For example, R30-30 would use a 30x30 ellipse for each corner. If w-h is omitted, 30-30 is used. R is valid only when W and H are present. Rectangle or ellipse: If the W and H options are present, the new display area will be a rectangle whose upper left corner is specified by the first (and only) pair of X-Y coordinates. However, if the E option is also present, the new display area will be an ellipse rather than a rectangle. For example: WinSetRegion "50-0 W200 H250 E", WinTitle. Polygon: When the W and H options are absent, the new display area will be a polygon determined by multiple pairs of X-Y coordinates (each pair of coordinates is a point inside the window relative to its upper left corner). For example, if three pairs of coordinates are specified, the new display area will be a triangle in most cases. The order of the coordinate pairs with respect to each other is sometimes important. In addition, the word Wind maybe be present in Options to use the winding method instead of the alternating method to determine the polygon's region. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. A ValueError is thrown if one or more Options are invalid, or if more than 2000 pairs of coordinates were specified. An OSError is thrown if the specified region is invalid or could not be applied to the target window. Remarks The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. When a region is set for a window owned by the script, the system may automatically change the method it uses to render the window's frame, thereby altering its appearance. The effect is similar to workaround #2 shown below, but only affects the window until its region is reset. Known limitation: Setting a region for a window not owned by the script may produce unexpected results if the window has a caption (title bar), and the system has desktop composition enabled. This is because the visible frame is not actually part of the window, but rendered by a separate system process known as the "desktop window manager". Note that desktop composition is always enabled on Windows 8 and later. One of the following two workarounds can be used: ; #1: Remove the window's caption. WinSetStyle "-0xC00000", "Window Title" ; To undo it: WinSetStyle "+0xC00000", "Window Title" ; #2: Disable DWM rendering of the window's frame. DllCall("dwmapi\DwmSetWindowAttribute", "ptr", WinExist("Window Title") , "uint", DWMWA_NCRENDERING_POLICY := 2, "int*", DWMNCRP_DISABLED := 1, "uint", 4) ; To undo it (this might also cause any set region to be ignored): DllCall("dwmapi\DwmSetWindowAttribute", "ptr", WinExist("Window Title") , "uint", DWMWA_NCRENDERING_POLICY := 2, "int*", DWMNCRP_ENABLED := 2, "uint", 4) Related Win functions, Control functions Examples Makes all parts of Notepad outside this rectangle invisible. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion "50-0 W200 H250", "ahk_class Notepad" Same as above but with corners rounded to 40x40. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion "50-0 W200 H250 R40-40", "ahk_class Notepad" Creates an ellipse instead of a rectangle. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion "50-0 W200 H250 E", "ahk_class Notepad" Creates a triangle with apex pointing down. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion "50-0 250-0 150-250", "ahk_class Notepad" Restores the window to its original/default display area. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion , "ahk_class Notepad" Creates a see-through rectangular hole inside Notepad (or any other window). There are two rectangles specified below: an outer and an inner. Each rectangle consists of 5 pairs of X/Y coordinates because the first pair is repeated at the end to "close off" each rectangle. This example may not work well with the new Notepad on Windows 11 or later. WinSetRegion "0-0 300-0 300-300 0-300 0-0 100-100 200-100 200-200 100-200 100-100", "ahk_class Notepad" WinSetStyle / WinSetExStyle - Syntax & Usage | AutoHotkey v2 WinSetStyle / WinSetExStyle Changes the style or extended style of the specified window, respectively. WinSetStyle Value , WinTitle, WinText, ExcludeTitle, ExcludeText WinSetExStyle Value , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Value Type: Integer or String Pass a positive integer to completely overwrite the window's style; that is, to set it to Value. To easily add, remove or toggle styles, pass a numeric string prefixed with a plus sign (+), minus sign (-) or caret (^), respectively. The new style value is calculated as shown below (where CurrentStyle could be retrieved with WinGetStyle/WinGetExStyle): Operation Prefix Example String Formula Add + +0x80 NewStyle := CurrentStyle | Value Remove - -0x80 NewStyle := CurrentStyle & ~Value Toggle ^ ^0x80 NewStyle := CurrentStyle ^ Value If Value is a negative integer, it is treated the same as the corresponding numeric string. To use the + or ^ prefix literally in an expression, the prefix or value must be enclosed in quote marks. For example: WinSetStyle("+0x80") or WinSetStyle("^" StylesToToggle). This is because the expression +123 produces 123 (without a prefix) and ^123 is a syntax error. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if the change could not be applied. Remarks See the styles table for a partial listing of styles. After applying certain style changes to a visible window, it might be necessary to redraw the window using WinRedraw. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinGetStyle / WinGetExStyle, ControlSetStyle / ControlSetExStyle, styles table, Win functions, Control functions Examples Removes the active window's title bar (WS_CAPTION). WinSetStyle "-0xC00000", "A" Toggles the WS_EX_TOOLWINDOW attribute, which removes/adds Notepad from the alt-tab list. WinSetExStyle "^0x80", "ahk_class Notepad" WinSetTitle - Syntax & Usage | AutoHotkey v2 WinSetTitle Changes the title of the specified window. WinSetTitle NewTitle , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters NewTitle Type: String The new title for the window. If this is the only parameter given, the Last Found Window will be used. WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if the change could not be applied. Remarks A change to a window's title might be merely temporary if the application that owns the window frequently changes the title. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinMove, WinGetTitle, WinGetText, ControlGetText, WinGetPos, Win functions Examples Changes the title of Notepad. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. WinSetTitle("This is a new title", "Untitled - Notepad") Opens Notepad, waits until it is active and changes its title. This example may fail on Windows 11 or later, as it requires the classic version of Notepad. Run "notepad.exe" WinWaitActive "Untitled - Notepad" WinSetTitle "This is a new title" ; Use the window found by WinWaitActive. Opens the main window, waits until it is active and changes its title. ListVars WinWaitActive "ahk_class AutoHotkey" WinSetTitle "This is a new title" ; Use the window found by WinWaitActive. WinSetTransColor - Syntax & Usage | AutoHotkey v2 WinSetTransColor Makes all pixels of the chosen color invisible inside the specified window. WinSetTransColor Color , WinTitle, WinText, ExcludeTitle, ExcludeText Parameters Color Type: String or Integer Specify a color name or RGB value (see the color chart for guidance, or use PixelGetColor in its RGB mode). To additionally make the visible part of the window partially transparent, append a space (not a

comma) followed by the transparency level (0-255). For example: WinSetTransColor "EEAA99 150", WinTitle. If the value is a string, any numeric color value must be in hexadecimal format. The color value can be omitted; for example, WinSetTransColor " 150" (with the leading space) is equivalent to WinSetTransparent 150. The word "Off" (case-insensitive) may be specified to completely turn off transparency for a window. This is functionally identical to WinSetTransparent "Off", WinTitle. Specifying "Off" is different than specifying 255 because it may improve performance and reduce usage of system resources (but probably only when desktop composition is disabled). WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if the change could not be applied. Remarks This allows the contents of the window behind it to show through. If the user clicks on an invisible pixel, the click will "fall through" to the window behind it. To change a window's existing TransColor, it may be necessary to turn off transparency before making the change. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. This function is often used to create on-screen displays and other visual effects. There is an example of an on-screen display at the bottom of the Gui object page. For a simple demonstration via hotkeys, see example #4 on the WinSetTransparent page. Related WinSetTransparent, Win functions, Control functions Examples Makes all white pixels in Notepad invisible. This example may not work well with the new Notepad on Windows 11 or later. WinSetTransColor "White", "Untitled - Notepad" WinSetTransparent - Syntax & Usage | AutoHotkey v2 WinSetTransparent Makes the specified window semi-transparent. WinSetTransparent N, WinTitle, WinText, ExcludeTitle, ExcludeText Parameters N Type: Integer or String To enable transparency, specify a number between 0 and 255 indicating the degree of transparency: 0 makes the window invisible while 255 makes it opaque. The word "Off" (case-insensitive) may be specified to completely turn off transparency for a window. This is functionally identical to WinSetTransColor "Off", WinTitle. Specifying "Off" is different than specifying 255 because it may improve performance and reduce usage of system resources (but probably only when desktop composition is disabled). WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found. An OSError is thrown if the change could not be applied. Remarks For example, to make the task bar transparent, use WinSetTransparent 150, "ahk_class Shell_TrayWnd". Similarly, to make the classic Start Menu transparent, see example #2. To make the Start Menu's submenus transparent, also include the script from example #3. Setting the transparency level to 255 before using "Off" might avoid window redrawing problems such as a black background. If the window still fails to be redrawn correctly, see WinRedraw for a possible workaround. The ID of the window under the mouse cursor can be retrieved with MouseGetPos. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinSetTransColor, Win functions, Control functions Examples Makes Notepad a little bit transparent. WinSetTransparent 200, "Untitled - Notepad" Makes the classic Start Menu transparent (to additionally make the Start Menu's submenus transparent, see example #3). DetectHiddenWindows True WinSetTransparent 150, "ahk_class BaseBar" Makes all or selected menus transparent throughout the system as soon as they appear. Note that although such a script cannot make its own menus transparent, it can make those of other scripts transparent. SetTimer WatchForMenu, 5 WatchForMenu() { DetectHiddenWindows True ; Might allow detection of menu sooner. if WinExist("ahk_class #32768") WinSetTransparent 150 ; Uses the window found by the above line. } Demonstrates the effects of WinSetTransparent and WinSetTransColor. Note: If you press one of the hotkeys while the mouse cursor is hovering over a pixel that is invisible as a result of TransColor, the window visible beneath that pixel will be acted upon instead! #t:: ; Press Win+T to make the color under the mouse cursor invisible. { MouseGetPos &MouseX, &MouseY, &MouseWin MouseRGB := PixelGetColor(MouseX, MouseY) ; It seems necessary to turn off any existing transparency first: WinSetTransColor "Off", MouseWin WinSetTransColor MouseRGB " 220", MouseWin } #o:: ; Press Win+O to turn off transparency for the window under the mouse. { MouseGetPos ,, &MouseWin WinSetTransColor "Off", MouseWin } #g:: ; Press Win+G to show the current settings of the window under the mouse. { MouseGetPos ,, &MouseWin Transparent := WinGetTransparent(MouseWin) TransColor := WinGetTransColor(MouseWin) ToolTip "Translucency:`t" Transparent "`nTransColor:`t" TransColor } WinShow - Syntax & Usage | AutoHotkey v2 WinShow Unhides the specified window. WinShow WinTitle, WinText, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Error Handling A TargetError is thrown if the window could not be found, except if the group mode is used. Remarks By default, WinShow is the only function that can always detect hidden windows. Other built-in functions can detect them only if DetectHiddenWindows has been turned on. This function operates only upon the topmost matching window except when WinTitle is ahk_group GroupName, in which case all windows in the group are affected. Related WinHide, SetTitleMatchMode, DetectHiddenWindows, Last Found Window Examples Opens Notepad, waits until it exists, hides it for a short time and unhides it. Run "notepad.exe" WinWait "Untitled - Notepad" Sleep 500 WinHide ; Use the window found by WinWait. Sleep 1000 WinShow ; Use the window found by WinWait. WinWait - Syntax & Usage | AutoHotkey v2 WinWait Waits until the specified window exists. WinWait WinTitle, WinText, Timeout, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinTitle may be blank only when WinText, ExcludeTitle, or ExcludeText is present. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. Timeout Type: Integer or Float How many seconds to wait before timing out and returning 0. Leave blank to allow the function to wait indefinitely. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer This function returns the HWND (unique ID) of a matching window if one was found, or 0 if the function timed out. Remarks If a matching window comes into existence, the function will not wait for Timeout to expire. Instead, it will update the Last Found Window and return, allowing the script to continue execution. If WinTitle specifies an invalid HWND (either as an Integer or via the Hwnd property of an object), the function returns immediately, without waiting for Timeout to expire. Waiting for another window to be created with the same HWND value would not be meaningful, as there would likely be no relation between the two windows. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. If another thread changes the contents of any variable(s) that were used for this function's parameters, the function will not see the change -- it will continue to use the title and text that were originally present in the variables when the function first started waiting. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on, even if WinTitle is a HWND or object. Related WinWaitActive, WinWaitClose, WinExist, WinActive, ProcessWait, SetTitleMatchMode, DetectHiddenWindows Examples Opens Notepad and waits a maximum of 3 seconds until it exists. If WinWait times out, an error message is shown, otherwise Notepad is minimized. Run "notepad.exe" if WinWait("Untitled - Notepad", , 3) WinMinimize ; Use the window found by WinWait. else MsgBox "WinWait timed out." WinWaitActive / WinWaitNotActive - Syntax & Usage | AutoHotkey v2 WinWaitActive / WinWaitNotActive Waits until the specified window is active or not active. WinWaitActive WinTitle, WinText, Timeout, ExcludeTitle, ExcludeText WinWaitNotActive WinTitle, WinText, Timeout, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text elements are detected if DetectHiddenText is ON. Timeout Type: Integer or Float How many seconds to wait before timing out and returning 0. Leave blank to allow the function to wait indefinitely. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer WinWaitActive returns the HWND (unique ID) of the active window if it matches the criteria, or 0 if the function timed out. WinWaitNotActive returns 1 if the active window does not match the criteria, or 0 if the function timed out. Remarks If the active window satisfies the function's expectation, the function will not wait for Timeout to expire. Instead, it will immediately return, allowing the script to resume. Since "A" matches whichever window is active at any given moment, WinWaitNotActive "A" typically waits indefinitely. To instead wait for a different window to become active, specify its unique ID as in the following example: WinWaitNotActive WinExist("A") Both WinWaitActive and WinWaitNotActive will update the Last Found Window if a matching window is active when the function begins or becomes active while the function is waiting. While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. If another thread changes the contents of any variable(s) that were used for this function's parameters, the function will not see the change -- it will continue to use the title and text that were originally present in the variables when the function first started waiting. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on. Related WinWait, WinWaitClose, WinExist, WinActive, SetTitleMatchMode, DetectHiddenWindows Examples Opens Notepad and waits a maximum of 2 seconds until it is active. If WinWait times out, an error message is shown, otherwise Notepad is minimized. Run "notepad.exe" if WinWaitActive("Untitled - Notepad", , 2) WinMinimize ; Use the window found by WinWaitActive. else MsgBox "WinWaitActive timed out." WinWaitClose - Syntax & Usage | AutoHotkey v2 WinWaitClose Waits until no matching windows can be found. WinWaitClose WinTitle, WinText, Timeout, ExcludeTitle, ExcludeText Parameters WinTitle Type: String, Integer or Object A window title or other criteria identifying the target window. See WinTitle. WinText Type: String If present, this parameter must be a substring from a single text element of the target window (as revealed by the included Window Spy utility). Hidden text

elements are detected if DetectHiddenText is ON. Timeout Type: Integer or Float How many seconds to wait before timing out and returning 0. Leave blank to allow the function to wait indefinitely. ExcludeTitle Type: String Windows whose titles include this value will not be considered. ExcludeText Type: String Windows whose text include this value will not be considered. Return Value Type: Integer (boolean) This function returns 0 (false) if the function timed out or 1 (true) otherwise. Remarks Whenever no matching windows exist, the function will not wait for Timeout to expire. Instead, it will immediately return 1 and the script will continue executing. Conversely, the function may continue waiting even after a matching window is closed, until no more matching windows can be found. Since "A" matches whichever window is active at any given moment, WinWaitClose "A" typically waits indefinitely. To instead wait for the current active window to close, specify its title or unique ID as in the following example: WinWaitClose WinExist("A") WinWaitClose updates the Last Found Window whenever it finds a matching window. One use for this is to identify or operate on the window after the function times out. For example: Gui("", "Test window " Random()).Show ("w300 h50") ; Show a test window. if !WinWaitClose("Test",, 5) ; Wait 5 seconds for someone to close it. { MsgBox "Window not yet closed: " WinGetTitle() WinClose ; Close the window found by WinWaitClose. } While the function is in a waiting state, new threads can be launched via hotkey, custom menu item, or timer. If another thread changes the contents of any variable(s) that were used for this function's parameters, the function will not see the change -- it will continue to use the title and text that were originally present in the variables when the function first started waiting. Window titles and text are case sensitive. Hidden windows are not detected unless DetectHiddenWindows has been turned on, even if WinTitle is a HWND or object. Related WinClose, WinWait, WinWaitActive, WinExist, WinActive, ProcessWaitClose, SetTitleMatchMode, DetectHiddenWindows Examples Opens Notepad, waits until it exists and then waits until it is closed. Run "notepad.exe" WinWait "Untitled - Notepad" WinWaitClose ; Use the window found by WinWait. MsgBox "Notepad is now closed." #ClipboardTimeout - Syntax & Usage | AutoHotkey v2 #ClipboardTimeout Changes how long the script keeps trying to access the clipboard when the first attempt fails. #ClipboardTimeout Milliseconds Parameters Milliseconds Type: Integer The length of the interval in milliseconds. Specify -1 to have it keep trying indefinitely. Specify 0 to have it try only once. Scripts that do not contain this directive use a 1000 ms timeout. Remarks Some applications keep the clipboard open for long periods of time, perhaps to write or read large amounts of data. In such cases, increasing this setting causes the script to wait longer before giving up and displaying an error message. This settings applies to all clipboard operations, the simplest of which are the following examples: Var := A_Clipboard and A_Clipboard := "New Text". Whenever the script is waiting for the clipboard to become available, new threads cannot be launched and timers will not run. However, if the user presses a hotkey, selects a custom menu item, or performs a GUI action such as pressing a button, that event will be buffered until later; in other words, its subroutine will be performed after the clipboard finally becomes available. This directive does not cause the reading of clipboard data to be reattempted if the first attempt fails. Like other directives, #ClipboardTimeout cannot be executed conditionally. Related A_Clipboard, Thread Examples Causes the script to wait 2 seconds instead of 1 second before giving up accessing the clipboard and displaying an error message. #ClipboardTimeout 2000 #DllLoad - Syntax & Usage | AutoHotkey v2 #DllLoad Loads a DLL or EXE file before the script starts executing. #DllLoad FileOrDirName Parameters FileOrDirName Type: String The path of a file or directory as explained below. This must not contain double quotes, wildcards or escape sequences other than semicolon (`;). Built-in variables may be used by enclosing them in percent signs (for example, #DllLoad %A_ScriptDir%). Percent signs which are not part of a valid variable reference are interpreted literally. All built-in variables are valid, except for A_Args and built-in classes. Known limitation: When compiling a script, variables are evaluated by the compiler and may differ from what the script would return when it is finally executed. The following variables are supported: A_AhkPath, A_AppData, A_AppDataCommon, A_ComputerName, A_ComSpec, A_Desktop, A_DesktopCommon, A_IsCompiled, A_LineFile, A_MyDocuments, A_ProgramFiles, A_Programs, A_ProgramsCommon, A_ScriptDir, A_ScriptFullPath, A_ScriptName, A_Space, A_StartMenu, A_StartMenuCommon, A_Startup, A_StartupCommon, A_Tab, A_Temp, A_UserName, A_WinDir. File: The absolute or relative path to the DLL or EXE file to be loaded. If a relative path is specified, the directive searches for the file using the same search strategy as the system's function LoadLibraryW. Note: SetWorkingDir has no effect on #DllLoad because #DllLoad is processed before the script begins executing. Directory: Specify a directory instead of a file to alter the search strategy by all subsequent occurrences of #DllLoad which do not specify an absolute path to a DLL or EXE. The new search strategy is the same as if Directory was passed to the system's function SetDllDirectoryW. If this parameter is omitted, the default search strategy is restored. Note: This parameter is not an expression, but can be enclosed in quote marks (either 'single' or "double"). Remarks Once a DLL or EXE has been loaded by this directive it cannot be unloaded by calling the system's function FreeLibrary. When the script is terminated, all loaded files are unloaded automatically. The FileName parameter may optionally begin with *i and a single space, which causes the program to ignore any failure to load the file. This option should be used only if the script is capable of executing despite the failure, such as if the DLL or EXE is non-essential, or if the script is designed to detect the failure. For example: #DllLoad "*i MyDLL" if !DllCall("GetModuleHandle", "str", "MyDLL") MsgBox "Failed to load MyDLL!" If the FileName parameter specifies a DLL name without a path and the file name extension is omitted, .dll is appended to the file name. To prevent this, include a trailing point character (.) in the file name. Like other directives, #DllLoad cannot be executed conditionally. Related DllCall Examples Loads a DLL file located in the current user's "My Documents" folder before the script starts executing. #DllLoad "%A_MyDocuments%\MyDLL.dll" #ErrorStdOut - Syntax & Usage | AutoHotkey v2 #ErrorStdOut Sends any syntax error that prevents a script from launching to the standard error stream (stderr) rather than displaying a dialog. #ErrorStdOut Encoding Parameters Encoding An encoding string specifying how to encode the output. For example, #ErrorStdOut "UTF-8" encodes error messages as UTF-8 before sending them to stderr. Whatever program is capturing the output must support UTF-8, and in some cases may need to be configured to expect it. If omitted, it defaults to CP0. Note: This parameter is not an expression, but can be enclosed in quote marks (either 'single' or "double"). Remarks Errors are written to stderr instead of stdout. The command prompt and fancy editors usually display both. This allows fancy editors such as TextPad, SciTE, Crimson, and EditPlus to jump to the offending line when a syntax error occurs. Since the #ErrorStdOut directive would have to be added to every script, it is usually better to set up your editor to use the command line switch /ErrorStdOut when launching any AutoHotkey script (see further below for setup instructions). Because AutoHotkey is not a console program, errors will not appear at the command prompt directly. This can be worked around by 1) compiling the script with the Ahk2Exe ConsoleApp directive, or 2) capturing the script's output via piping or redirection. For example: "C:\Program Files\AutoHotkey\AutoHotkey.exe" /ErrorStdOut "My Script.ahk" 2>&1 |more "C:\Program Files\AutoHotkey\AutoHotkey.exe" /ErrorStdOut "My Script.ahk" 2>"Syntax-Error Log.txt" You can also pipe the output directly to the clipboard by using the operating system's built-in clip command. For example: "C:\Program Files\AutoHotkey\AutoHotkey.exe" /ErrorStdOut "My Script.ahk" 2>&1 |clip Note: 2>&1 causes stderr to be redirected to stdout, while 2>Filename redirects only stderr to a file. Like other directives, #ErrorStdOut cannot be executed conditionally. Instructions for specific editors EditPlus: From the menu bar, select Tools > Configure User Tools. Press button: Add Tool > Program Menu Text: Your choice Command: C:\Program Files\AutoHotkey\AutoHotkey.exe Argument: /ErrorStdOut "$(FilePath)" Initial directory: $(FileDir) Capture output: Yes TextPad: From the menu bar, select Configure > Preferences. Expand the Tools entry. Press the Add button and select "Program". Copy and paste (adjust to your path): C:\Windows\System32\cmd.exe -- then press OK. Triple-click the newly added item (cmd.exe) in the ListBox and rename it to your choice (e.g. Launch Script). Press Apply. Select the new item in the tree at the left and enter the following information: Command (should already be filled in): cmd.exe (or the full path to it) Parameters (adjust to your path, if necessary): /c ""C:\Program Files\AutoHotkey\AutoHotkey.exe" /ErrorStdOut "$File"" Initial folder: $FileDir Check the following boxes: 1) Run minimized; 2) Capture output. Press OK. The newly added item should now exist in the Tools menu. Related FileAppend (because it can also send text to stderr or stdout) Examples Sends any syntax error that prevents the script from launching to stderr rather than displaying a dialog. #ErrorStdOut #HotIf - Syntax & Usage | AutoHotkey v2 #HotIf Creates context-sensitive hotkeys and hotstrings. Such hotkeys perform a different action (or none at all) depending on any condition (an expression). #HotIf Expression Parameters Expression Type: Boolean Any valid expression. This becomes the return value of an implicit function which has one parameter (ThisHotkey). The function cannot modify global variables directly (as it is assume-local as usual, and cannot contain declarations), but can call other functions which do. Basic Operation The #HotIf directive sets the expression which will be used by subsequently created hotkeys to determine whether they should activate. This expression is evaluated when the key, mouse button or combination is pressed, or at other times when the program needs to know whether the hotkey is active. To make context-sensitive hotkeys and hotstrings, simply precede them with the #HotIf directive. For example: #HotIf WinActive("ahk_class Notepad") or WinActive(MyWindowTitle) #Space::MsgBox "You pressed Win+Spacebar in Notepad or " MyWindowTitle The #HotIf directive is positional: it affects all hotkeys and hotstrings physically beneath it in the script, until the next #HotIf directive. Note: Unlike if statements, braces have no effect with the #HotIf directive. To turn off context sensitivity, specify #HotIf without any expression. For example: #HotIf Like other directives, #HotIf cannot be executed conditionally. When a mouse or keyboard hotkey is disabled via #HotIf, it performs its native function; that is, it passes through to the active window as though there is no such hotkey. There is one exception: Joystick hotkeys: although #HotIf works, it never prevents other programs from seeing the press of a button. #HotIf can also be used to alter the behavior of an ordinary key like Enter or Space. This is useful when a particular window ignores that key or performs some action you find undesirable. For example: #HotIf WinActive("Reminders ahk_class #32770") ; The "reminders" window in Outlook. Enter::Send "!o" ; Have an "Enter" keystroke open the selected reminder rather than snoozing it. #HotIf Variant (Duplicate) Hotkeys A particular hotkey or hotstring can be defined more than once in the script if each definition has different #HotIf criteria. These are known as hotkey variants. For example: #HotIf WinActive("ahk_class Notepad") ^!c::MsgBox "You pressed Control+Alt+C in Notepad." #HotIf WinActive("ahk_class WordPadClass") ^!c::MsgBox "You pressed Control+Alt+C in WordPad." #HotIf ^!c::MsgBox "You pressed Control+Alt+C in a window other than Notepad/WordPad." If more than one variant is eligible to fire, only the one closest to the top of the script will fire. The exception to this is the global variant (the one with no #HotIf criteria): It always has the lowest precedence; therefore, it will fire only if no other variant is eligible (this exception does not apply to hotstrings). When creating duplicate hotkeys, the order of modifier symbols such as ^!+# does not matter. For example, ^!c is the same as !^c. However, keys must be spelled consistently. For example, Esc is not the same as Escape for this purpose (though the case does not matter). Also, any hotkey with a wildcard prefix (*) is entirely separate from a non-wildcard one; for example, *F1 and F1 would each have their own

set of variants. A window group can be used to make a hotkey execute for a group of windows. For example: GroupAdd "MyGroup", "ahk_class Notepad" GroupAdd "MyGroup", "ahk_class WordPadClass" #HotIf WinActive("ahk_group MyGroup") #z::MsgBox "You pressed Win+Z in either Notepad or WordPad." To create hotkey variants dynamically (while the script is running), see HotIf. Expression Evaluation When the key, mouse or joystick button combination which forms a hotkey is pressed, the #HotIf expression is evaluated to determine if the hotkey should activate. Note: Scripts should not assume that the expression is only evaluated when the key is pressed (see below). The expression may also be evaluated whenever the program needs to know whether the hotkey is active. For example, the #HotIf expression for a custom combination like a & b:: might be evaluated when the prefix key (a in this example) is pressed, to determine whether it should act as a custom modifier key. Note: Use of #HotIf in an unresponsive script may cause input lag or break hotkeys (see below). There are several more caveats to the #HotIf directive: Keyboard or mouse input is typically buffered (delayed) until expression evaluation completes or times out. Expression evaluation can only be performed by the script's main thread (at the OS level, not a quasi-thread), not directly by the keyboard/mouse hook. If the script is busy or unresponsive, such as if a FileCopy is in progress, expression evaluation is delayed and may time out. If the system-defined timeout is reached, the system may stop notifying the script of keyboard or mouse input (see #HotIfTimeout for details). Sending keystrokes or mouse clicks while the expression is being evaluated (such as from a function which it calls) may cause complications and should be avoided. ThisHotkey, A_ThisHotkey and A_TimeSinceThisHotkey are set based on the hotkey for which the current #HotIf expression is being evaluated. A_PriorHotkey and A_TimeSincePriorHotkey temporarily contain the previous values of the corresponding "This" variables. Optimization #HotIf is optimized to avoid expression evaluation for simple calls to WinActive or WinExist, thereby reducing the risk of lag or other issues in such cases. Specifically: The expression must contain exactly one call to WinExist or WinActive. Each parameter must be a single quoted string, and no more than two parameters may be used. The result may be inverted with not or !, but no other operators may be used. Whitespace and parentheses are fully handled when the expression is pre-compiled and therefore do not affect this optimization. If the expression meets these criteria, it is evaluated directly by the program and does not appear in ListLines. Before the Hotkey function is used to modify an existing hotkey variant, typically the HotIf function must be used with the original expression text. However, the first unique expression with a given combination of criteria can also be referenced by that criteria. For example: HotIfWinExist "ahk_class Notepad" Hotkey "#n", "Off" ; Turn the hotkey off. HotIf 'WinExist("ahk_class Notepad")' Hotkey "#n", "On" ; Turn the same hotkey back on. #HotIf WinExist("ahk_class Notepad") #n::WinActivate Note that any use of variables will disqualify the expression. If the variable's value never changes after the hotkey is created, there are two strategies for minimizing the risk of lag or other issues inherent to #HotIf: Use HotIfWin... MyTitleVar to set the criteria and Hotkey KeyName, Label to create the hotkey variant. Use a constant expression such as #HotIf WinActive("ahk_group MyGroup") and define the window group with GroupAdd "MyGroup", MyTitleVar elsewhere in the script. General Remarks #HotIf also restores prefix keys to their native function when appropriate (a prefix key is A in a hotkey such as a & b). This occurs whenever there are no enabled hotkeys for a given prefix. When a hotkey is currently disabled via #HotIf, its key or mouse button will appear with a "#" character in KeyHistory's "Type" column. This can help debug a script. Alt-tab hotkeys are not affected by #HotIf: they are in effect for all windows. The Last Found Window can be set by #HotIf. For example: #HotIf WinExist("ahk_class Notepad") #n::WinActivate ; Activates the window found by WinExist(). Related #HotIfTimeout may be used to override the default timeout value. Hotkey function, Hotkeys, Hotstrings, Suspend, WinActive, WinExist, SetTitleMatchMode, DetectHiddenWindows Examples Creates two hotkeys and one hotstring which only work when Notepad is active, and one hotkey which works for any window except Notepad. #HotIf WinActive("ahk_class Notepad") ^!a::MsgBox "You pressed Ctrl-Alt-A while Notepad is active." #c::MsgBox "You pressed Win-C while Notepad is active." ::btw::This replacement text for "btw" will occur only in Notepad. #HotIf #c::MsgBox "You pressed Win-C in a window other than Notepad." Allows the volume to be adjusted by scrolling the mouse wheel over the taskbar. #HotIf MouseIsOver("ahk_class Shell_TrayWnd") WheelUp::Send "{Volume_Up}" WheelDown::Send "{Volume_Down}" MouseIsOver(WinTitle) { MouseGetPos ,, &Win return WinExist(WinTitle " ahk_id " Win) } Simple word-delete shortcuts for all Edit controls. #HotIf ActiveControlIsOfClass("Edit") ^BS::Send "^+{Left}{Del}" ^Del::Send "^+{Right}{Del}" ActiveControlIsOfClass(Cls) { FocusedControl := 0 try FocusedControl := ControlGetFocus("A") FocusedControlClass := "" try FocusedControlClass := WinGetClass(FocusedControl) return (FocusedControlClass=Cls) } Context-insensitive Hotkey. #HotIf Esc::ExitApp Dynamic Hotkeys. This example should be combined with example #2 before running it. NumpadAdd:: { static toggle := false HotIf 'MouseIsOver("ahk_class Shell_TrayWnd")' if (toggle := !toggle) Hotkey "WheelUp", DoubleUp else Hotkey "WheelUp", "WheelUp" return ; Nested function: DoubleUp(ThisHotkey) => Send("{Volume_Up 2}") } #HotIfTimeout - Syntax & Usage | AutoHotkey v2 #HotIfTimeout Sets the maximum time that may be spent evaluating a single #HotIf expression. #HotIfTimeout Timeout Parameters Timeout Type: Integer The timeout value to apply globally, in milliseconds. Remarks A timeout is implemented to prevent long-running expressions from stalling keyboard input processing. If the timeout value is exceeded, the expression continues to evaluate, but the keyboard hook continues as if the expression had already returned false. If this directive is unspecified in the script, it will behave as though set to 1000. Note that the system implements its own timeout, defined by the DWORD value LowLevelHooksTimeout in the following registry key: HKEY_CURRENT_USER\Control Panel\Desktop If the system timeout value is exceeded, the system may stop calling the script's keyboard hook, thereby preventing hook hotkeys from working until the hook is re-registered or the script is reloaded. The hook can usually be re-registered by suspending and un-suspending all hotkeys. Microsoft's documentation is unclear about the details of this timeout, but research indicates the following for Windows 7 and later: If LowLevelHooksTimeout is not defined, the default timeout is 300ms. The hook may time out up to 10 times, but is silently removed if it times out an 11th time. If a given hotkey has multiple #HotIf variants, the timeout might be applied to each variant independently, making it more likely that the system timeout will be exceeded. This may be changed in a future update. Like other directives, #HotIfTimeout cannot be executed conditionally. Related #HotIf Examples Sets the #HotIf timeout to 10 ms instead of 1000 ms. #HotIfTimeout 10 #Hotstring - Syntax & Usage | AutoHotkey v2 #Hotstring Changes hotstring options or ending characters. #Hotstring NoMouse #Hotstring EndChars NewChars #Hotstring NewOptions Parameters NoMouse Type: String Prevents mouse clicks from resetting the hotstring recognizer as described here. As a side-effect, this also prevents the mouse hook from being required by hotstrings (though it will still be installed if the script requires it for other purposes, such as mouse hotkeys). The presence of #Hotstring NoMouse anywhere in the script affects all hotstrings, not just those physically beneath it. EndChars NewChars Type: String Specify the word EndChars followed a single space and then the new ending characters. For example: #Hotstring EndChars -()[]{}':;"/\,.?!`n`s`t Since the new ending characters are in effect globally for the entire script -- regardless of where the EndChars directive appears -- there is no need to specify EndChars more than once. The maximum number of ending characters is 100. Characters beyond this length are ignored. To make tab or space an ending character, include `t or `s in the list. NewOptions Type: String Specify new options as described in Hotstring Options. For example: #Hotstring r s k0 c0. Unlike EndChars above, the #Hotstring directive is positional when used this way. In other words, entire sections of hotstrings can have different default options as in this example: ::btw::by the way #Hotstring r c ; All the below hotstrings will use "send raw" and will be case sensitive by default. ::al::airline ::CEO::Chief Executive Officer #Hotstring c0 ; Make all hotstrings below this point case insensitive. Remarks Like other directives, #Hotstring cannot be executed conditionally. Related Hotstrings The Hotstring function can be used to change hotstring options while the script is running. #Include / #IncludeAgain - Syntax & Usage | AutoHotkey v2 #Include / #IncludeAgain Causes the script to behave as though the specified file's contents are present at this exact position. #Include FileOrDirName #Include #IncludeAgain FileOrDirName Parameters FileOrDirName Type: String The path of a file or directory as explained below. This must not contain double quotes (except for an optional pair of double quotes surrounding the parameter), wildcards or escape sequences other than semicolon (`;). Built-in variables may be used by enclosing them in percent signs (for example, #Include "%A_ScriptDir%"). Percent signs which are not part of a valid variable reference are interpreted literally. All built-in variables are valid, except for A_Args and built-in classes. Known limitation: When compiling a script, variables are evaluated by the compiler and may differ from what the script would return when it is finally executed. The following variables are supported: A_AhkPath, A_AppData, A_AppDataCommon, A_ComputerName, A_ComSpec, A_Desktop, A_DesktopCommon, A_IsCompiled, A_LineFile, A_MyDocuments, A_ProgramFiles, A_Programs, A_ProgramsCommon, A_ScriptDir, A_ScriptFullPath, A_ScriptName, A_Space, A_StartMenu, A_StartMenuCommon, A_Startup, A_StartupCommon, A_Tab, A_Temp, A_UserName, A_WinDir. File: The name of the file to be included. By default, relative paths are relative to the directory of the file which contains the #Include directive. This default can be overridden by using #Include Dir as described below. Note: SetWorkingDir has no effect on #Include because #Include is processed before the script begins executing. Directory: Specify a directory instead of a file to change the working directory used by all subsequent occurrences of #Include and FileInstall in the current file. Note: Changing the working directory in this way does not affect the script's initial working directory when it starts running (A_WorkingDir). To change that, use SetWorkingDir at the top of the script. Note: This parameter is not an expression, but can be enclosed in quote marks (either 'single' or "double"). LibName Type: String A library file or function name. For example, #include and #include would both include lib.ahk from one of the Lib folders. LibName cannot contain variable references. Remarks A script behaves as though the included file's contents are physically present at the exact position of the #Include directive (as though a copy-and-paste were done from the included file). Consequently, it generally cannot merge two isolated scripts together into one functioning script. #Include ensures that FileName is included only once, even if multiple inclusions are encountered for it. By contrast, #IncludeAgain allows multiple inclusions of the same file, while being the same as #Include in all other respects. The FileName parameter may optionally be preceded by *i and a single space, which causes the program to ignore any failure to read the included file. For example: #Include "*i SpecialOptions.ahk". This option should be used only when the included file's contents are not essential to the main script's operation. Lines displayed in the main window via ListLines or the menu View->Lines are always numbered according to their physical order within their own files. In other words, including a new file will change the line numbering of the main script file by only one line, namely that of the #Include line itself (except for compiled scripts, which merge their included files into one big script at the time of compilation). #Include is often used to load functions defined in an external file. Like other directives, #Include cannot be executed conditionally. In other words, this example would not work as expected: if (x = 1) #Include "SomeFile.ahk" ; This line takes effect regardless of

the value of x. Related Script Library Folders, Functions, FileInstall Examples Includes the contents of the specified file into the current script. #Include "C:\My Documents\Scripts\Utility Subroutines.ahk" Changes the working directory for subsequent #Includes and FileInstalls. #Include "%A_ScriptDir%" Same as above but for an explicitly named directory. #Include "C:\My Scripts" #InputLevel - Syntax & Usage | AutoHotkey v2 #InputLevel Controls which artificial keyboard and mouse events are ignored by hotkeys and hotstrings. #InputLevel Level Parameters Level Type: Integer An integer between 0 and 100. If omitted, it defaults to 0. General Remarks For an explanation of how SendLevel and #InputLevel are used, see SendLevel. This directive is positional: it affects all hotkeys and hotstrings between it and the next #InputLevel directive. If not specified by an #InputLevel directive, hotkeys and hotstrings default to level 0. A hotkey's input level can also be set using the Hotkey function. For example: Hotkey "#z", my_hotkey_sub, "I1" The input level of a hotkey or non-auto-replace hotstring is also used as the default send level for any keystrokes or button clicks generated by that hotkey or hotstring. Since a keyboard or mouse remapping is actually a pair of hotkeys, this allows #InputLevel to be used to allow remappings to trigger other hotkeys. AutoHotkey versions older than v1.1.06 behave as though #InputLevel 0 and SendLevel 0 are in effect. Like other directives, #InputLevel cannot be executed conditionally. Related SendLevel, Hotkeys, Hotstrings Examples Causes the first hotkey *Numpad0:: to trigger the second hotkey ~LButton::. This would be not the case if the #InputLevel directives are omitted or commented out. #InputLevel 1 ; Use SendEvent so that the script's own hotkeys can be triggered. *Numpad0::SendEvent "{Blind}{Click Down}" *Numpad0 up::SendEvent "{Blind}{Click Up}" #InputLevel 0 ; This hotkey can be triggered by both Numpad0 and LButton: ~LButton::MsgBox "Clicked" #MaxThreads - Syntax & Usage | AutoHotkey v2 #MaxThreads Sets the maximum number of simultaneous threads. #MaxThreads Value Parameters Value Type: Integer The maximum total number of threads that can exist simultaneously. Specifying a number higher than 255 is the same as specifying 255. Remarks This setting is global, meaning that it needs to be specified only once (anywhere in the script) to affect the behavior of the entire script. Although a value of 1 is allowed, it is not recommended because it would prevent new hotkeys from launching whenever the script is displaying a message box or other dialog. It would also prevent timers from running whenever another thread is sleeping or waiting. The OnExit callback function will always launch regardless of how many threads exist. If this setting is lower than #MaxThreadsPerHotkey, it effectively overrides that setting. If this directive is unspecified in the script, it will behave as though set to 10. Like other directives, #MaxThreads cannot be executed conditionally. Related #MaxThreadsPerHotkey, Threads, A_MaxHotkeysPerInterval, ListHotkeys Examples Allows a maximum of 2 instead of 10 simultaneous threads. #MaxThreads 2 #MaxThreadsBuffer - Syntax & Usage | AutoHotkey v2 #MaxThreadsBuffer Causes some or all hotkeys to buffer rather than ignore keypresses when their #MaxThreadsPerHotkey limit has been reached. #MaxThreadsBuffer Setting Parameters Setting Specify one of the following literal values (if omitted, it defaults to True): True or 1: All hotkey subroutines between here and the next #MaxThreadsBuffer False directive will buffer rather than ignore presses of their hotkeys whenever their subroutines are at their #MaxThreadsPerHotkey limit. False or 0: This is the default behavior. A hotkey press will be ignored whenever that hotkey is already running its maximum number of threads (usually 1, but this can be changed with #MaxThreadsPerHotkey). Remarks This directive is rarely used because this type of buffering, which is OFF by default, usually does more harm than good. For example, if you accidentally press a hotkey twice, having this setting ON would cause that hotkey's subroutine to automatically run a second time if its first thread takes less than 1 second to finish (this type of buffer expires after 1 second, by design). Note that AutoHotkey buffers hotkeys in several other ways (such as Thread "Interrupt" and Critical). It's just that this particular way can be detrimental, thus it is OFF by default. The main use for this directive is to increase the responsiveness of the keyboard's auto-repeat feature. For example, when you hold down a hotkey whose #MaxThreadsPerHotkey setting is 1 (the default), incoming keypresses are ignored if that hotkey subroutine is already running. Thus, when the subroutine finishes, it must wait for the next auto-repeat keypress to come in, which might take 50ms or more due to being caught in between keystrokes of the auto-repeat cycle. This 50ms delay can be avoided by enabling this directive for any hotkey that needs the best possible response time while it is being auto-repeated. As with all # directives, this one should not be positioned in the script as though it were a function (i.e. it is not necessary to have it contained within a subroutine). Instead, position it immediately before the first hotkey you wish to have affected by it. Like other directives, #MaxThreadsBuffer cannot be executed conditionally. Related #MaxThreads, #MaxThreadsPerHotkey, Critical, Thread (function), Threads, Hotkey, A_MaxHotkeysPerInterval, ListHotkeys Examples Causes the first two hotkeys to buffer rather than ignore keypresses when their #MaxThreadsPerHotkey limit has been reached. #MaxThreadsBuffer True #x::MsgBox "This hotkey will use this type of buffering." #y::MsgBox "And this one too." #MaxThreadsBuffer False #z::MsgBox "But not this one." #MaxThreadsPerHotkey - Syntax & Usage | AutoHotkey v2 #MaxThreadsPerHotkey Sets the maximum number of simultaneous threads per hotkey or hotstring. #MaxThreadsPerHotkey Value Parameters Value Type: Integer The maximum number of threads that can be launched for a given hotkey/hotstring subroutine (limit 255). Remarks This setting is used to control how many "instances" of a given hotkey or hotstring subroutine are allowed to exist simultaneously. For example, if a hotkey has a max of 1 and it is pressed again while its subroutine is already running, the press will be ignored. This is helpful to prevent accidental double-presses. However, if you wish these keypresses to be buffered rather than ignored -- perhaps to increase the responsiveness of the keyboard's auto-repeat feature -- use #MaxThreadsBuffer. Unlike #MaxThreads, this setting is not global. Instead, position it before the first hotkey you wish to have affected by it, which will result in all subsequent hotkeys using that value until another instance of this directive is encountered. The setting of #MaxThreads -- if lower than this setting -- takes precedence. If this directive is unspecified in the script, it will behave as though set to 1. Like other directives, #MaxThreadsPerHotkey cannot be executed conditionally. Related #MaxThreads, #MaxThreadsBuffer, Critical, Threads, Hotkey, A_MaxHotkeysPerInterval, ListHotkeys Examples Allows a maximum of 3 simultaneous threads instead of 1 per hotkey or hotstring. #MaxThreadsPerHotkey 3 #NoTrayIcon - Syntax & Usage | AutoHotkey v2 #NoTrayIcon Disables the showing of a tray icon. #NoTrayIcon Specifying this anywhere in a script will prevent the showing of a tray icon for that script when it is launched (even if the script is compiled into an EXE). If you use this for a script that has hotkeys, you might want to bind a hotkey to the ExitApp function. Otherwise, there will be no easy way to exit the program (without restarting the computer or killing the process). For example: #x::ExitApp. The tray icon can be made to disappear or reappear at any time during the execution of the script by assigning a true or false value to A_IconHidden. The only drawback of using A_IconHidden := true at the very top of the script is that the tray icon might be briefly visible when the script is first launched. To avoid that, use #NoTrayIcon instead. The built-in variable A_IconHidden contains 1 if the tray icon is currently hidden or 0 otherwise, and can be assigned a value to show or hide the icon. Like other directives, #NoTrayIcon cannot be executed conditionally. Related Tray Icon, TraySetIcon, A_IconHidden, A_IconTip, ExitApp Examples Causes the script to launch without a tray icon. #NoTrayIcon #Requires - Syntax & Usage | AutoHotkey v2 #Requires Displays an error and quits if a version requirement is not met. #Requires Requirement Parameters Requirement If this does not begin with the word "AutoHotkey", an error message is shown and the program exits. This encourages clarity and reserves the directive for future uses. Other forks of AutoHotkey may support other names. Otherwise, the word "AutoHotkey" should be followed by any combination of the following, separated by spaces or tabs: An optional letter "v" followed by a version number. A_AhkVersion is required to be greater than or equal to this version, but less than the next major version. One of <, <=, >, >= or = immediately followed by an optional letter "v" and a version number. For example, >=2-rc <2 allows v2 release candidates but not the final release. One of the following words to restrict the type of executable (EXE) which can run the script: "32-bit", "64-bit". Error Message The message shown depends on the version of AutoHotkey interpreting the directive. For v2, the path, version and build of AutoHotkey are always shown in the error message. If the script is launched with a version of AutoHotkey that does not support this directive, the error message is something like the following: Line Text: #Requires %Requirement% Error: This line does not contain a recognized action. Remarks If the script uses syntax or functions which are unavailable in earlier versions, using this directive ensures that the error message shows the unmet requirement, rather than indicating an arbitrary syntax error. This cannot be done with something like if (A_AhkVersion <= "1.1.33") because a syntax error elsewhere in the script would prevent it from executing. When sharing a script or posting code online, using this directive allows anyone who finds the code to readily identify which version of AutoHotkey it was intended for. Other programs or scripts can check for this directive for various purposes. For example, the launcher installed with AutoHotkey v2 uses it to determine which AutoHotkey executable to launch, while a script editor or related tools might use it to determine how to interpret or highlight the script file. Version strings are compared as a series of dot-delimited components, optionally followed by a hyphen and pre-release identifier(s). Numeric components are compared numerically. For example, v1.01 = v1.1, but a20 > a112. Numeric components are always considered lower than non-numeric components in the same position. Any missing dot-delimited components are assumed to be zero. For example, v1.1.33-alpha is the same as v1.1.33.00-alpha.0. Non-numeric components are compared alphabetically, and are case sensitive. Pre-release versions are considered lower than standard releases. For example, a script that #Requires AutoHotkey v2 will not run on v2.0-a112. To permit pre-release versions, include a hyphen suffix. For example: v2.0-. Any suffix beginning with "+" is ignored. A trailing "+" is sufficient to indicate to the reader that later versions are acceptable, but is not required. Like other directives, #Requires cannot be executed conditionally. Related VerCompare, #ErrorStdOut Examples Causes the script to run only on v2.0, including alpha releases. #Requires AutoHotkey v2.0-a MsgBox "This script will run only on v2.0, including alpha releases." Causes the script to run only on v2.0, including pre-release versions. #Requires AutoHotkey >=2.0- <2.1 Causes the script to run only with a 64-bit interpreter (EXE). #Requires AutoHotkey 64-bit Causes the script to run only with a 64-bit interpreter (EXE) version 2.0-rc.2 or later. #Requires AutoHotkey v2.0-rc.2 64-bit #SingleInstance - Syntax & Usage | AutoHotkey v2 #SingleInstance Determines whether a script is allowed to run again when it is already running. #SingleInstance ForceIgnorePromptOff Parameters ForceIgnorePromptOff Type: String If the parameter is omitted, it defaults to Force. To change this behavior, specify one of the following words: Force: Skips the dialog box and replaces the old instance automatically, which is similar in effect to the Reload function. Ignore: Skips the dialog box and leaves the old instance running. In other words, attempts to launch an already-running script are ignored. Prompt: Displays a dialog box asking whether to keep the old instance or replace it with the new one. This is the default behaviour if this directive is not used. Off: Allows multiple instances of the script to run concurrently. Remarks This directive is ignored when any of the following command line switches are used: /force /restart Like other directives, #SingleInstance cannot be executed conditionally. Limitations Previous instances of the script are identified by searching for a main window with the default title. Therefore, a previous instance may

not be found if: The title of its main window has been changed. It is running on a different version of AutoHotkey. Its main window is no longer top-level, such as if the script has used SetParent to change its parent to something other than NULL (0). At most one previous instance is detected and sent a message asking it to close. Therefore, the following additional limitations also apply: If there are multiple instances (such as if previous instances of the script used the #SingleInstance Off mode), the topmost matching instance is sent the message, and other instances are not considered. If the previous instance is running at a higher integrity level than the new instance (where running as administrator > running with UI access > normal), it cannot be closed due to security restrictions. If multiple instances of the script are started simultaneously, they may fail to detect each other or may all target the same previous instance. This would result in multiple instances of the script starting. Related Reload Examples Skips the dialog box and replaces the old instance automatically. #SingleInstance Force #SuspendExempt - Syntax & Usage | AutoHotkey v2 #SuspendExempt Exempts subsequent hotkeys and hotstrings from suspension. #SuspendExempt Exempt Parameters Exempt True or 1 to enable exemption for subsequent hotkeys, or False or 0 to disable exemption. If omitted, it defaults to True. Remarks Hotkeys and hotstrings can be suspended by the Suspend function or via the tray icon or main window. If this directive is unspecified in the script, all hotkeys or hotstrings are disabled when the script is suspended, even those which call the Suspend function. This directive does not affect the Hotkey or Hotstring functions, for which the S hotkey option or S hotstring option can be used instead. Like other directives, #SuspendExempt cannot be executed conditionally. Related Suspend, Hotkeys, Hotstrings Examples The first hotkey in this example toggles the suspension. To prevent this hotkey from being suspended after the suspension has been turned on and thus no longer being able to turn it off, it must be exempted. #SuspendExempt ; Exempt the following hotkey from Suspend. #Esc::Suspend -1 #SuspendExempt False ; Disable exemption for any hotkeys/hotstrings below this. ^1::MsgBox "This hotkey is affected by Suspend." #UseHook - Syntax & Usage | AutoHotkey v2 #UseHook Forces the use of the hook to implement all or some keyboard hotkeys. #UseHook Setting Parameters Setting Specify one of the following literal values (if omitted, it defaults to True): True or 1: The keyboard hook will be used to implement all keyboard hotkeys between here and the next #UseHook false (if any). False or 0: Hotkeys will be implemented using the default method (RegisterHotkey() if possible; otherwise, the keyboard hook). Remarks Normally, the windows API function RegisterHotkey () is used to implement a keyboard hotkey whenever possible. However, the responsiveness of hotkeys might be better under some conditions if the keyboard hook is used instead. Turning this directive ON is equivalent to using the $ prefix in the definition of each affected hotkey. As with all # directives -- which are processed only once when the script is launched -- #UseHook should not be positioned in the script as though it were a function (that is, it is not necessary to have it contained within a subroutine). Instead, position it immediately before the first hotkey you wish to have affected by it. By default, hotkeys that use the keyboard hook cannot be triggered by means of the Send function. Similarly, mouse hotkeys cannot be triggered by built-in functions such as Click because all mouse hotkeys use the mouse hook. One workaround is to name the hotkey's function and call it directly. #InputLevel and SendLevel provide additional control over which hotkeys and hotstrings are triggered by the Send function. Like other directives, #UseHook cannot be executed conditionally. Related InstallKeybdHook, InstallMouseHook, ListHotkeys, #InputLevel Examples Causes the first two hotkeys to use the keyboard hook. #UseHook ; Force the use of the hook for hotkeys after this point. #x::MsgBox "This hotkey will be implemented with the hook." #y::MsgBox "And this one too." #UseHook False #z::MsgBox "But not this one." #Warn - Syntax & Usage | AutoHotkey v2 #Warn Enables or disables warnings for specific conditions which may indicate an error, such as a typo or missing "global" declaration. #Warn WarningType, WarningMode Parameters WarningType Type: String The type of warning to enable or disable. If omitted, it defaults to All. VarUnset: Before the script starts to run, display a warning for the first reference to each variable which is never used in any of the following ways: As the target of a direct, non-dynamic assignment such as MyVar := "". Used with the reference operator (e.g. &MyVar). Passed directly to IsSet (e.g. IsSet(MyVar)). LocalSameAsGlobal: Before the script starts to run, display a warning for each undeclared local variable which has the same name as a global variable. This is intended to prevent errors caused by forgetting to declare a global variable inside a function before attempting to assign to it. If the variable really was intended to be local, a declaration such as local x or static y can be used to suppress the warning. This warning is disabled by default. #Warn g := 1 ShowG() { ; The warning is displayed even if the function is never called. ;global g ; <-- This is required to assign to the global variable. g := 2 } ShowG MsgBox g ; Without the declaration, the above assigned to a local "g". Unreachable: Before the script starts to run, show a warning for each line that immediately follows a Return, Break, Continue, Throw or Goto at the same nesting level, unless that line is the target of a label. Any such line would never be executed. If the code is intended to be unreachable - such as if a return has been used to temporarily disable a block of code, or a hotkey or hotstring has been temporarily disabled by commenting it out - consider commenting out the unreachable code as well. Alternatively, the warning can be suppressed by defining a label above the first unreachable line. All: Apply the given WarningMode to all supported warning types. WarningMode Type: String A value indicating how warnings should be delivered. If omitted, it defaults to MsgBox. MsgBox: Show a message box describing the warning. Note that once the message box is dismissed, the script will continue as usual. StdOut: Send a description of the warning to stdout (the program's standard output stream), along with the filename and line number. This allows fancy editors such as SciTE to capture warnings without disrupting the script - the user can later jump to each offending line via the editor's output pane. OutputDebug: Send a description of the warning to the debugger for display. If a debugger is not active, this will have no effect. For more details, see OutputDebug. Off: Disable warnings of the given WarningType. Remarks By default, all warnings are enabled and use the MsgBox mode, except for LocalSameAsGlobal, which is disabled. The checks which produce VarUnset, LocalSameAsGlobal and Unreachable warnings are performed after all directives have been parsed, but before the script executes. Therefore, the location in the script is not significant (and, like other directives, #Warn cannot be executed conditionally). However, the ordering of multiple #Warn directives is significant: the last occurrence that sets a given warning determines the mode for that warning. So, for example, the two statements below have the combined effect of enabling all warnings except LocalSameAsGlobal: #Warn All #Warn LocalSameAsGlobal, Off Related Local and Global Variables Examples Disables all warnings. Not recommended. #Warn All, Off Enables every type of warning and shows each warning in a message box. #Warn Sends a warning to OutputDebug for each undeclared local variable which has the same name as a global variable. #Warn LocalSameAsGlobal, OutputDebug #WinActivateForce - Syntax & Usage | AutoHotkey v2 #WinActivateForce Skips the gentle method of activating a window and goes straight to the forceful method. #WinActivateForce Specifying this anywhere in a script will cause built-in functions that activate a window -- such as WinActivate, WinActivateBottom, and GroupActivate -- to skip the "gentle" method of activating a window and go straight to the more forceful methods. Although this directive will usually not change how quickly or reliably a window is activated, it might prevent task bar buttons from flashing when different windows are activated quickly one after the other. Remarks Like other directives, #WinActivateForce cannot be executed conditionally. Related WinActivate, WinActivateBottom, GroupActivate Examples Enables the forceful method of activating a window. #WinActivateForce Acknowledgements | AutoHotkey v2 Acknowledgements In addition to the AutoIt authors already mentioned: Robert Yaklin: A lot of tireless testing to isolate bugs, a great first draft of the installer, as well as great suggestions for how commands ought to work :) Jason Payam Ahdoot: For suggesting and describing floating point support. Jay D. Novak: For discovering many Win9x problems with the Send command, CapsLock, and hotkey modifiers; and for generously sharing his wealth of code and wisdom for hotkeys, hot-strings, hook usage, and typing acceleration. Rajat: For creating stylish replacements for the original AHK icons; a great product logo; making the syntax customizations for TextPad; discovering some bugs with the registry commands and AutoIt v2 compatibility; making SmartGUI Creator; and many other things. Thaddeus Beck (Beardboy): For NT4 testing to fix GetKeyState and the Send command; for a lot of help on the forum; and for many suggestions and bug reports. Gregory F. Hogg of Hogg's Software: For writing the source code for multi-monitor support in the SysGet command. Aurelian Maga: For writing the source code for ImageSearch and the faster PixelSearch. Joost Mulders: Whose source code provided the foundation for expressions. Laszlo Hars: For advice about data structures and algorithms, which helped greatly speed up arrays and dynamic variables. Marcus Sonntag (Ultra): For the research, design, coding, and testing of DllCall. Gena Shimanovich: For debugging and coding assistance; and for some advanced prototype scripts upon which future features may be based. Eric Morin (numEric): For advice on mathematics; for steadfast debugging of areas like OnMessage, floating point computations, and mouse movement; and for improvements to overall quality. Philip Hazel: For Perl-Compatible Regular Expressions (PCRE). Titan/polyethene: For providing community hosting on autohotkey.net, creating many useful scripts and libraries, creating the JavaScript to colorize code-comments in the forum, and many other things. Philippe Lhoste (PhiLho): For tireless moderation and support in the forum, RegEx advice and testing, syntax and design ideas, and many other things. John Biederman: For greatly improving the presentation and ergonomics of the documentation. Jonathan Rennison (JGR): For developing RegisterCallback (now called CallbackCreate), and for beneficial suggestions. Steve Gray (Lexikos): For developing dynamic function calling and other new functionality; analyzing and fixing bugs; and valuable support and advice for hundreds of individual visitors at the forum. AutoHotkey_L: jackeiku: For developing Unicode support and other new functionality. fincs: For developing native 64-bit support and try/catch/throw, writing a replacement for the old compiler, analyzing and fixing bugs. Sean: For developing built-in COM functionality and providing valuable insight into COM. TheGood: For merging the documentation and adapting the installer script. ac: For developing #Warn. Russell Davis: For developing A_PriorKey, ahk_path (the basis of ahk_exe in WinTitle parameters), #InputLevel and SendLevel. Christian Sander: For developing support for SysLink controls. And to everyone else who's contributed or sent in bug reports or suggestions: Thanks! Script Compiler Directives | AutoHotkey v2 Script Compiler Directives Table of Contents Introduction Directives that control the script behaviour: IgnoreBegin IgnoreEnd IgnoreKeep Directives that control executable metadata: Introduction AddResource: Adds a resource to the .exe. Bin / Base: Specifies the base version of AutoHotkey to use. ConsoleApp: Sets Console mode. Cont: Specifies a directive continuation line. Debug: Shows directive debugging text. ExeName: Specifies the location and name for the .exe. Let: Sets a user variable. Nop: Does nothing. Obey: Obeys a command or expression. PostExec: Runs a program after compilation. ResourceID: Assigns a non-standard resource ID to the main script. SetMainIcon: Sets the main icon. SetProp: Sets an .exe property. Set: Sets a miscellaneous property. UpdateManifest: Changes the .exe's manifest. UseResourceLang: Changes the resource language. Introduction Script compiler directives allow the user to specify details of how a script is to be compiled via Ahk2Exe. 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. The script compiler looks for special comments in the source script and recognises these as Compiler Directives. All compiler directives are introduced by the string @Ahk2Exe-, preceded by the comment flag (usually ;). Directives that control the script behaviour It is possible to remove code sections from the compiled script by wrapping them in directives: MsgBox "This message appears in both the compiled and uncompiled script" ;@Ahk2Exe-IgnoreBegin MsgBox "This message does NOT appear in the compiled script" ;@Ahk2Exe-IgnoreEnd MsgBox "This message appears in both the compiled and uncompiled script" The reverse is also possible, i.e. marking a code section to only be executed in the compiled script: /*@Ahk2Exe-Keep MsgBox "This message appears only in the compiled script" */ MsgBox "This message appears in both the compiled and uncompiled script" This has advantage over A_IsCompiled because the code is completely removed from the compiled script during preprocessing, thus making the compiled script smaller. The reverse is also true: it will not be necessary to check for A_IsCompiled because the code is inside a comment block in the uncompiled script. Directives that control executable metadata Introduction In the parameters of these directives, the following escape sequences are supported: ``, `,, `n, `r and `t. Commas always need to be escaped, regardless of the parameter position. "Integer" refers to unsigned 16-bit integers (0..0xFFFF). If required, directive parameters can reference the following list of standard built-in variables by enclosing the variable name with % signs: Group 1: A_AhkPath, A_AppData, A_AppDataCommon, A_ComputerName, A_ComSpec, A_Desktop, A_DesktopCommon, A_MyDocuments, A_ProgramFiles, A_Programs, A_ProgramsCommon, A_ScriptDir, A_ScriptFullPath, A_ScriptName, A_Space, A_StartMenu, A_StartMenuCommon, A_Startup, A_StartupCommon, A_Tab, A_Temp, A_UserName, A_WinDir. Group 2: A_AhkVersion, A_IsCompiled, A_PtrSize. In addition to these variable names, the special variable A_WorkFileName holds the temporary name of the processed .exe file. This can be used to pass the file name as a parameter to any PostExec directives which need to access the generated .exe. Furthermore, the special variable A_BasePath contains the full path and name of the selected base file. Also, the special variable A_PriorLine contains the source line immediately preceding the current compiler directive. Intervening lines of blanks and comments only are ignored, as are any intervening compiler directive lines. This variable can be used to 'pluck' constant information from the script source, and use it in later compiler directives. An example would be accessing the version number of the script, which may be changed often. Accessing the version number in this way means that it needs to be changed only once in the source code, and the change will get copied through to the necessary directive. (See the RegEx example below for more information.) As well, special user variables can be created with the format U_Name using the Let and Obey directives, described below. In addition to being available for directive parameters, all variables can be accessed from any RT_MENU, RT_DIALOG, RT_STRING, RT_ACCELERATORS, RT_HTML, and RT_MANIFEST file supplied to the AddResource directive, below. If needed, the value returned from the above variables can be manipulated by including at the end of the built-in variable name before the ending %, up to 2 parameters (called p2 and p3) all separated by tilde ~. The p2 and p3 parameters will be used as literals in the 2nd and 3rd parameters of a RegExReplace function to manipulate the value returned. (See RegEx Quick Reference.) Note that p3 is optional. To include a tilde as data in p2 or p3, preceded it with a back-tick, i.e. `~. To include a back-tick character as data in p2 or p3, double it, i.e. ``. RegEx examples: % A_ScriptName~\.[^\.]+$~.exe% This replaces the extension plus preceding full-stop, with .exe in the actual script name.(\.[^\.]+$~.exe means scan for a . followed by 1 or more non-. characters followed by end-of-string, and replace them with .exe) Assume there is a source line followed by two compiler directives as follows: CodeVersion := "1.2.3.4", company := "My Company" ;@Ahk2Exe-Let U_version = %A_PriorLine~U)^(.+"){1}(.+)".*$~$2% ;@Ahk2Exe-Let U_company = % A_PriorLine~U)^(.+"){3}(.+)".*$~$2% These directives copy the version number 1.2.3.4 into the special variable U_version, and the company name My Company into the special variable U_company for use in other directives later. (The {1} in the first regex was changed to {3} in the second regex to select after the third " to extract the company name.) Other examples: Other working examples which can be downloaded and examined, are available from here. AddResource Adds a resource to the compiled executable. (Also see UseResourceLang below) ;@Ahk2Exe-AddResource FileName , ResourceName FileName The filename of the resource to add. The file is assumed to be in (or relative to) the script's own directory if an absolute path isn't specified. The type of the resource (as an integer or string) can be explicitly specified by prepending an asterisk to it: *type Filename. If omitted, Ahk2Exe automatically detects the type according to the file extension. ResourceName (Optional) The name that the resource will have (can be a string or an integer). If omitted, it defaults to the name (with no path) of the file, in uppercase. Here is a list of common standard resource types and the extensions that trigger them by default. 2 (RT_BITMAP): .bmp, .dib 4 (RT_MENU) 5 (RT_DIALOG) 6 (RT_STRING) 9 (RT_ACCELERATORS) 10 (RT_RCDATA): Every single other extension. 11 (RT_MESSAGETABLE) 12 (RT_GROUP_CURSOR): .cur (not yet supported) 14 (RT_GROUP_ICON): .ico 23 (RT_HTML): .htm, .html, .mht 24 (RT_MANIFEST): .manifest. If the name for the resource is not specified, it defaults to 1 Example 1: To replace the standard icons (other than the main icon): ;@Ahk2Exe-AddResource Icon1.ico, 160 ; Replaces 'H on blue' ;@Ahk2Exe-AddResource Icon2.ico, 206 ; Replaces 'S on green' ;@Ahk2Exe-AddResource Icon3.ico, 207 ; Replaces 'H on red' ;@Ahk2Exe-AddResource Icon4.ico, 208 ; Replaces 'S on red' Example 2: To include another script as a separate RCDATA resource (see Embedded Scripts): ;@Ahk2Exe-AddResource MyScript1.ahk, #2 ;@Ahk2Exe-AddResource MyScript2.ahk, MYRESOURCE Note that each script added with this directive will be fully and separately processed by the compiler, and can include further directives. If there are any competing directives overall, the last encountered by the compiler will be used. Bin / Base Specifies the base version of AutoHotkey to be used to generate the .exe file. This directive may be overridden by a base file parameter specified in the GUI or CLI. This directive can be specified many times if necessary, but only in the top level script file (i.e. not in an #Include file). The compiler will be run at least once for each Bin/Base directive found. (If an actual comment is appended to this directive, it must use the ; flag. To truly comment out this directive, insert a space after the first comment flag.) ;@Ahk2Exe-Bin [Path\]Name , [Exe_path\][Name], Codepage ; Deprecated ;@Ahk2Exe-Base [Path\]Name , [Exe_path\][Name], Codepage [Path\]Name The *.bin or *.exe file to use. If no extension is supplied, .bin is assumed. The file is assumed to be in (or relative to) the compiler's own directory if an absolute path isn't specified. A DOS mask may be specified for Name, e.g. ANSI*, Unicode 32*, Unicode 64*, or *bit for all three. The compiler will be run for each *.bin or *.exe file that matches. Any use of built-in variable replacements must only be from group 1 above. [Exe_path\][Name] (Optional) The file name to be given to the .exe. Any extension supplied will be replaced by .exe. If no path is specified, the .exe will be created in the script folder. If no name is specified, the .exe will have the default name. (This parameter can be overridden by the ExeName directive.) Codepage (Optional) Overrides the default codepage used to process script files. (Scripts should begin with a Unicode byte-order-mark (BOM), rendering the use of this parameter unnecessary.) ConsoleApp Changes the executable subsystem to Console mode. ;@Ahk2Exe-ConsoleApp Cont Specifies a continuation line for the preceding directive. This allows a long-lined directive to be formatted so that it is easy to read in the source code. ;@Ahk2Exe-Cont Text Text The text to be appended to the previous directive line, before that line is processed. The text starts after the single space following the Cont key-word. Debug Shows a message box with the supplied text, for debugging purposes. ;@Ahk2Exe-Debug Text Text The text to be shown. Include any special variables between % signs to see the (manipulated) contents. ExeName Specifies the location and name given to the generated .exe file. (Also see the Base directive.) This directive may be overridden by an output file specified in the GUI or CLI. ;@Ahk2Exe-ExeName [Path\][Name] [Path\][Name] The .exe file name. Any extension supplied will be replaced by .exe. If no path is specified, the .exe will be created in the script folder. If no name is specified, the .exe will have the default name. Example: ;@Ahk2Exe-Obey U_bits, = %A_PtrSize% * 8 ;@Ahk2Exe-Obey U_type, = "%A_IsUnicode%" ? "Unicode" : "ANSI" ;@Ahk2Exe-ExeName %A_ScriptName~\.[^\.]+$%_%U_type%_%U_bits% Let Creates (or modifies) one or more user variables which can be accessed by %U_Name%, similar to the built-in variables (see above). ;@Ahk2Exe-Let Name = Value , Name = Value, ... Name The name of the variable (with or without the leading U_). Value The value to be used. Nop Does nothing. ;@Ahk2Exe-Nop Text Text (Optional) Any text, which is ignored. Example: Ver := A_AhkVersion "" ; If quoted literal not empty, do 'SetVersion' ;@Ahk2Exe-Obey U_V, = "%A_PriorLine~U)^(.+")(.*)".*$~$2%" ? "SetVersion" : "Nop" ;@Ahk2Exe-%U_V% %A_AhkVersion%%A_PriorLine~U)^(.+")(.*)".*$~$2% Obey Obeys isolated AutoHotkey commands or expressions, with result in U_Name. ;@Ahk2Exe-Obey Name, CmdOrExp , Extra Name The name of the variable (with or without the leading U_) to receive the result. CmdOrExp The command or expression to obey. Command format must use Name as the output variable (often the first parameter), e.g. ;@Ahk2Exe-Obey U_date, FormatTime U_date`, R D2 T2 Expression format must start with =, e.g. ;@Ahk2Exe-Obey U_type, = "%A_IsUnicode%" ? "Unicode" : "ANSI" Expressions can be written in command format, e.g. ;@Ahk2Exe-Obey U_bits, U_bits := %A_PtrSize% * 8 If needed, separate multiple commands and expressions with `n. Extra (Optional) A number (1-9) specifying the number of extra results to be returned. e.g. if extra = 2, results will be returned in U_name, U_name1, and U_name2. The values in the names must first be set by the expression or command. PostExec Specifies a program to be executed after a successful compilation, before (or after) any Compression is applied to the .exe. This directive can be present many times and will be executed in the order encountered by the compiler, in the appropriate queue as specified by the When parameter. ;@Ahk2Exe-PostExec Program [parameters] , When, WorkingDir, Hidden, IgnoreErrors Program [parameters] The program to execute, plus parameters. To allow access to the processed .exe file, specify the special variable A_WorkFileName as a quoted parameter, such as "%A_WorkFileName%". If the program changes the .exe, the altered .exe must be moved back to the input file specified by % A_WorkFileName%, by the program. (Note that the .exe will contain binary data.) When (Optional) Leave blank to execute before any Compression is done. Otherwise set to a number to run after compression as follows: 0 - Only run when no compression is specified. 1 - Only run when MPRESS compression is specified. 2 - Only run when UPX compression is specified. WorkingDir (Optional) The working directory for the program. Do not enclose the name in double quotes even if it contains spaces. If omitted, the directory of the compiler (Ahk2Exe) will be used. Hidden (Optional) If set to 1, the program will be launched hidden. IgnoreErrors (Optional) If set to 1, any errors that occur during the launching or running of the program will not be reported to the user. Example 1: (To use the first two examples, first download BinMod.ahk and compile it according to the instructions in the downloaded script.) This example can be used to remove a reference to "AutoHotkey" in the generated .exe to disguise that it is a compiled AutoHotkey script: ;@Ahk2Exe-Obey U_au, = "%A_IsUnicode%" ? 2 : 1 ; Script ANSI or Unicode? ;@Ahk2Exe-PostExec "BinMod.exe" "%A_WorkFileName%" ;@Ahk2Exe-Cont "%U_au%2.>AUTOHOTKEY SCRIPT<. DATA " Example 2: This example will alter a UPX compressed .exe so that it can't be de-compressed with UPX -d: ;@Ahk2Exe-PostExec "BinMod.exe" "%

A_WorkFileName%" ;@Ahk2Exe-Cont "11.UPX." "1.UPX!.", 2 (There are other examples mentioned in the BinMod.ahk script.) Example 3: This example specifies the Compression to be used on a compiled script, if none is specified in the CLI or GUI. The default parameters normally used by the compiler are shown.For MPRESS: ;@Ahk2Exe-PostExec "MPRESS.exe" "%A_WorkFileName%" -q -x, 0,, 1 For UPX: ;@Ahk2Exe-PostExec "UPX.exe" "% A_WorkFileName%" ;@Ahk2Exe-Cont -q --all-methods --compress-icons=0, 0,, 1 ResourceID Assigns a non-standard resource ID to be used for the main script for compilations which use an .exe base file (see Embedded Scripts). This directive may be overridden by a Resource ID specified in the GUI or CLI. This directive is ignored if it appears in a script inserted by the AddResource directive. ;@Ahk2Exe-ResourceID Name Name The resource ID to use. Numeric resource IDs should consist of a hash sign (#) followed by a decimal number. SetMainIcon Overrides the custom EXE icon used for compilation. (To change the other icons, see the AddResource example.) This directive may be overridden by an icon file specified in the GUI or CLI. The new icon might not be immediately visible in Windows Explorer if the compiled file existed before with a different icon, however the new icon can be shown by selecting Refresh Windows Icons from the Ahk2Exe File menu. ;@Ahk2Exe-SetMainIcon IcoFile IcoFile (Optional) The icon file to use. If omitted, the default AutoHotkey icon is used. SetProp Changes a property in the compiled executable's version information. Note that all properties are processed in alphabetical order, regardless of the order they are specified. ;@Ahk2Exe-SetProp Value Prop The name of the property to change. Must be one of those listed below. Property Description CompanyName Changes the company name. Copyright Changes the legal copyright information. Description Changes the file description. On Windows 8 and above, this also changes the script's name in Task Manager under "Processes". FileVersion Changes the file version, in both text and raw binary format. (See Version below, for more details.) InternalName Changes the internal name. Language Changes the language code. Please note that hexadecimal numbers must have an 0x prefix. LegalTrademarks Changes the legal trademarks information. Name Changes the product name and the internal name. OrigFilename Changes the original filename information. ProductName Changes the product name. ProductVersion Changes the product version, in both text and raw binary format. (See Version below, for more details.) Version Changes the file version and the product version, in both text and raw binary format. Ahk2Exe fills the binary version fields with the period-delimited numbers (up to four) that may appear at the beginning of the version text. Unfilled fields are set to zero. For example, 1.3-alpha would produce a binary version number of 1.3.0.0. If this property is not modified, it defaults to the AutoHotkey version used to compile the script. Value The value to set the property to. Set Changes other miscellaneous properties in the compiled executable's version information not covered by the SetProp directive. Note that all properties are processed in alphabetical order, regardless of the order they are specified. This directive is for specialised use only. ;@Ahk2Exe-Set Prop, Value Prop The name of the property to change. Value The value to set the property to. UpdateManifest Changes details in the .exe's manifest. This directive is for specialised use only. ;@Ahk2Exe-UpdateManifest RequireAdmin , Name, Version, UIAccess RequireAdmin Set to 1 to change the executable to require administrative privileges when run. Set to 2 to change the executable to request highest available privileges when run. Set to 0 to leave unchanged. Name (Optional) The name to be set in the manifest. Version (Optional) The version to be set in the manifest. UIAccess (Optional) Set to 1 to make UIAccess true in the manifest. UseResourceLang Changes the resource language used by AddResource. This directive is positional and affects all AddResource directives that follow it. ;@Ahk2Exe-UseResourceLang LangCode LangCode The language code. Please note that hexadecimal numbers must have an 0x prefix. The default resource language is US English (0x0409). CLSID List (Windows Class Identifiers) | AutoHotkey v2 CLSID List (Windows Class Identifiers) Certain special folders within the operating system are identified by unique strings. Some of these strings can be used with FileSelect, DirSelect, and Run. For example: OutputVar := FileSelect(, "::{645ff040-5081-101b-9f08-00aa002f954e}") ; Select a file in the Recycle Bin. OutputVar := DirSelect("::{20d04fe0-3aea-1069-a2d8-08002b30309d}") ; Select a folder within My Computer. CLSID Location Run? ::{d20ea4e1-3957-11d2-a40b-0c5020524153} Administrative Tools ::{85bbd920-42a0-1069-a2e4-08002b30309d} Briefcase ::{21ec2020-3aea-1069-a2dd-08002b30309d} Control Panel ::{d20ea4e1-3957-11d2-a40b-0c5020524152} Fonts ::{ff393560-c2a7-11cf-bff4-444553540000} History ::{00020d75-0000-0000-c000-000000000046} Inbox ::{00028b00-0000-0000-c000-000000000046} Microsoft Network ::{20d04fe0-3aea-1069-a2d8-08002b30309d} My Computer Yes ::{450d8fba-ad25-11d0-98a8-0800361b1103} My Documents Yes ::{208d2c60-3aea-1069-a2d7-08002b30309d} My Network Places Yes ::{1f4de370-d627-11d1-ba4f-00a0c91eedba} Network Computers Yes ::{7007acc7-3202-11d1-aad2-00805fc1270e} Network Connections Yes ::{2227a280-3aea-1069-a2de-08002b30309d} Printers and Faxes Yes ::{7be9d83c-a729-4d97-b5a7-1b7313c39e0a} Programs Folder ::{645ff040-5081-101b-9f08-00aa002f954e} Recycle Bin Yes ::{e211b736-43fd-11d1-9efb-0000f8757fcd} Scanners and Cameras ::{d6277990-4c6a-11cf-8d87-00aa0060f5bf} Scheduled Tasks Yes ::{48e7caab-b918-4e58-a94d-505519c795dc} Start Menu Folder ::{7bd29e00-76c1-11cf-9dd0-00a0c9034933} Temporary Internet Files ::{bdeadf00-c265-11d0-bced-00a0c90ab50f} Web Folders The "Yes" entries in the last column are not authoritative: the Run function might support different CLSIDs depending on system configuration. To open a CLSID folder via Run, simply specify the CLSID as the first parameter. For example: Run "::{20d04fe0-3aea-1069-a2d8-08002b30309d}" ; Opens the "My Computer" folder. Run "::{645ff040-5081-101b-9f08-00aa002f954e}" ; Opens the Recycle Bin. Run "::{450d8fba-ad25-11d0-98a8-0800361b1103}\My Folder" ; Opens a folder that exists inside "My Documents". Run A_MyDocuments "\My Folder" ; Same as the above on most systems. List of Color Names | AutoHotkey v2 Color names and RGB values Color name RGB value Black 000000 Silver C0C0C0 Gray 808080 White FFFFFF Maroon 800000 Red FF0000 Purple 800080 Fuchsia FF00FF Green 008000 Lime 00FF00 Olive 808000 Yellow FFFF00 Navy 000080 Blue 0000FF Teal 008080 Aqua 00FFFF DPI Scaling | AutoHotkey v2 DPI Scaling DPI scaling is a function performed either by the operating system or by the application, to increase the visual size of content proportionate to the "dots per inch" setting of the display. Generally it allows content to appear at the same physical size on systems with different display resolutions, or to at least be usable on very high-resolution displays. Sometimes a user may increase the DPI setting just to make content larger and more comfortable to read. A_ScreenDPI returns the DPI setting of the primary screen. There are two types of DPI scaling that relate to AutoHotkey: Gui DPI Scaling and OS DPI Scaling. Gui DPI Scaling Automatic scaling is performed by the Gui and GuiControl methods/properties by default, so that GUI scripts with hard-coded positions, sizes and margins will tend to scale as appropriate on high DPI screens. If this interferes with the script, or if the script will do its own scaling, the automatic scaling can be disabled. For more details, see the -DPIScale option. OS DPI Scaling For applications which are not DPI-aware, the operating system automatically scales coordinates passed to and returned from certain system functions. This type of scaling affects AutoHotkey only on systems with multiple screens where not all screens have the same DPI setting. Per-Monitor DPI Awareness On Windows 8.1 and later, secondary screens can have different DPI settings, and "per-monitor DPI-aware" applications are expected to scale their windows according to the DPI of whichever screen they are currently on, adapting dynamically when the window moves between screens. For applications which are not per-monitor DPI-aware, the system performs bitmap scaling to allow windows to change sizes when they move between screens, and hides this from the application by reporting coordinates and sizes scaled to the global DPI setting that the application expects to have. For instance, on an 11 inch 4K screen, a GUI designed to display at 96 dpi (100%) would be almost impossible to use, whereas upscaling it by 200% would make it usable. AutoHotkey is not designed to perform per-monitor scaling, and therefore has not been marked as per-monitor DPI-aware. This is a boon, for instance, when moving a GUI window between a large external screen with 100% DPI and a smaller screen with 200% DPI. However, automatic scaling does have negative implications. In order of the system's automatic scaling to work, system functions such as MoveWindow and GetWindowRect automatically scale the coordinates that they accept or return. When AutoHotkey uses these functions to work with external windows, this often produces unexpected results if the coordinates are not on the primary screen. To add further confusion, some functions scale coordinates based on which screen the script's last active window was displayed on. Workarounds On Windows 10 version 1607 and later, the SetThreadDpiAwarenessContext system function can be used to change the program's DPI awareness setting at runtime. For instance, enabling per-monitor DPI awareness disables the scaling performed by the system, so built-in functions such as WinMove and WinGetPos will accept or return coordinates in pixels, untouched by DPI scaling. However, if a GUI is sized for a screen with 100% DPI and then moved to a screen with 200% DPI, it will not adjust automatically, and may be very hard to use. To enable per-monitor DPI awareness, call the following function prior to using functions that are normally affected by DPI scaling: DllCall("SetThreadDpiAwarenessContext", "ptr", -3, "ptr") On Windows 10 version 1703 and later, -3 can be replaced with -4 to enable the "Per Monitor v2" mode. This enables scaling of dialogs, menus, tooltips and some other things. However, it also causes the non-client area (title bar) to scale, which may cause the window's client area to be too small unless the script is designed to adjust for it (such as by responding to the WM_DPICHANGED message). This can be avoided by setting the context to -3 before creating the GUI, but -4 before creating any tooltips, menus or dialogs. The thread's DPI awareness may temporarily change while the user is moving one of the script's windows, or while the script is displaying a dialog. Therefore, it is safest to set the DPI awareness immediately before using any functions which rely on it. Compiled Scripts Per-monitor DPI awareness can be enabled process-wide by changing the content of the element of the compiled script's manifest resource from true (the default set in the base AutoHotkey executable file) to true/pm. Editors with AutoHotkey Support | AutoHotkey v2 Editors with AutoHotkey Support Any text editor can be used to edit an AutoHotkey script, but editors which are (or can be configured to be) more AutoHotkey-aware tend to make reading, editing and testing scripts much easier. AutoHotkey-aware editors may provide: Syntax highlighting, like what is used in this documentation. With syntax highlighting, words, symbols and segments of code are colour-coded to indicate their meaning. For instance, literal text, comments and variable names are displayed in different colours. Auto-complete, which typically offers a list of suggestions when you start typing the name of a known function or variable. Calltips may show you what the parameters are for a function while you are writing code to call the function. Interactive debugging, such as to step through the script line by line and inspect variables at each step, or to view and modify variables or objects, beyond what ListVars allows. Recommendations: SciTE4AutoHotkey is simple to install, relatively light-weight, and fully supports AutoHotkey v1 and v2 without further configuration. VS Code (plus extensions) provides an even higher level of support and a wider range of features, but can be heavy on resources. SciTE4AutoHotkey SciTE4AutoHotkey is a custom version of the text editor known as SciTE. Its features include: Syntax highlighting Auto-complete Calltips Smart auto-indent Code folding Interactive debugging Running the script by pressing a hotkey Other tools for AutoHotkey scripting SciTE4AutoHotkey can be found here: https://www.autohotkey.com/scite4ahk/ Visual Studio Code (VS Code) Visual Studio Code can be configured with a high level of support for AutoHotkey by installing extensions. AutoHotkey2 Language Support provides

many features, including: Syntax highlighting Auto-complete Calltips Smart auto-indent Code folding Running the script by pressing a hotkey Real-time diagnostics (detecting common errors) Formatting/tidying code Additional notes: This extension only supports AutoHotkey v2, but can also detect v1 scripts and automatically switch over to a v1 extension if one is installed. This extension can also be used with other editors, such as vim, neovim and Sublime Text 4. For details, see Use in other editors. However, VS Code likely provides the best experience, with easiest setup. vscode-autohotkey-debug provides support for interactive debugging of v1 and v2 scripts. Notepad++ Notepad++ can be configured to support the following features: Syntax highlighting Auto-complete Code folding Running the script by pressing a hotkey See Setup Notepad++ for AutoHotkey for instructions. Notepad2-zufuliu Notepad2-zufuliu supports the following for AutoHotkey v2 by default: Syntax highlighting Auto-complete Auto-indent Code folding Running the script by pressing a hotkey It is available here: https://github.com/zufuliu/notepad2 Others editors For help finding or configuring other editors, try the Editors sub-forum. To get an editor added to this page, post in the Suggestions sub-forum or open an Issue or Pull Request at GitHub. Escape Sequences - Definition & Usage | AutoHotkey v2 Escape Sequences The escape character ` (back-tick or grave accent) is used to indicate that the character immediately following it should be interpreted differently than it normally would. This character is at the upper left corner of most English keyboards. In AutoHotkey the following escape sequences can be used: Sequence Result `` ` (literal accent; i.e. two consecutive escape characters result in a single literal character) `; ; (literal semicolon) Note: It is not necessary to escape a semicolon which has any character other than space or tab to its immediate left, since it would not be interpreted as a comment anyway. `: : (literal colon). This is necessary only in a hotstring's triggering abbreviation. `{ { (keyboard key). This is only valid, and is required, when remapping a key to {. `n newline (linefeed/LF) `r carriage return (CR) `b backspace `t tab (the more typical horizontal variety) `s space `v vertical tab -- corresponds to Ascii value 11. It can also be manifest in some applications by typing Ctrl+K. `a alert (bell) -- corresponds to Ascii value 7. It can also be manifest in some applications by typing Ctrl+G. `f formfeed -- corresponds to Ascii value 12. It can also be manifest in some applications by typing Ctrl+L. `" or `' 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: Var := "The color `"red`" was found." or Var := 'The color "red" was found.'. Examples Reports a multi-line string. The lines are separated by a linefeed character. MsgBox "Line 1`nLine 2" Standard Windows Fonts | AutoHotkey v2 Standard Windows Fonts For use with Gui.SetFont. The column "Exists Since" represents the first Windows version in which the font was included. Common monospaced (fixed-width) fonts in this table are Consolas, Courier, Courier New, Fixedsys, Lucida Console and Terminal. The availability of some fonts may depend on the language of the operating system. Recommend Fonts are highlighted in yellow. Note that fonts recommended here are fonts which are readable and have been included since Windows Vista or earlier. Using such fonts ensures that they are displayed on most operating systems. By default, the table is sorted by font names in ascending order. To sort the table by another column or in another order, click on a column header. Font Name Mainly Designed For Exists Since Aharoni Hebrew 2000 Aldhabi Arabic 8 Andalus Arabic 2000 Angsana New Thai XP AngsanaUPC Thai XP Aparajita Indic 7 Arabic Typesetting Arabic Vista Arial Latin, Greek, Cyrillic 95 Arial Black Latin, Greek, Cyrillic 98 Arial Nova Latin, Greek, Cyrillic 10 Bahnschrift Latin 10 Batang Korean 2000 BatangChe Korean Vista Browallia New Thai 2000 BrowalliaUPC Thai 2000 Calibri Latin, Greek, Cyrillic Vista Calibri Light Latin, Greek, Cyrillic 8 Cambria Latin, Greek, Cyrillic Vista Cambria Math Symbols Vista Candara Latin, Greek, Cyrillic Vista Comic Sans MS Latin, Greek, Cyrillic 95 Consolas Latin, Greek, Cyrillic Vista Constantia Latin, Greek, Cyrillic Vista Corbel Latin, Greek, Cyrillic Vista Cordia New Thai 2000 CordiaUPC Thai 2000 Courier Latin, Greek, Cyrillic 95 Courier New Latin, Greek, Cyrillic 95 DaunPenh Other Vista David Hebrew 2000 DengXian Chinese 10 DFKai-SB Chinese 2000 DilleniaUPC Thai 2000 DokChampa Other Vista Dotum Korean Vista DotumChe Korean Vista Ebrima African 7 Estrangelo Edessa Other XP EucrosiaUPC Thai 2000 Euphemia Other Vista FangSong Chinese Vista Fixedsys Latin 95 Franklin Gothic Medium Latin, Greek, Cyrillic XP FrankRuehl Hebrew 2000 FreesiaUPC Thai 2000 Gabriola Latin, Greek, Cyrillic 7 Gadugi Other 8 Gautami Other XP Georgia Latin, Greek, Cyrillic 2000 Georgia Pro Latin, Greek, Cyrillic 10 Gill Sans Nova Latin, Greek, Cyrillic 10 Gisha Hebrew Vista Gulim Korean 2000 GulimChe Korean Vista Gungsuh Korean Vista GungsuhChe Korean Vista HoloLens MDL2 Assets Symbols 10 Impact Latin, Greek, Cyrillic 98 IrisUPC Thai 2000 Iskoola Pota Other Vista JasmineUPC Thai 2000 Javanese Text Javanese 8.1 KaiTi Chinese 2000 Kalinga Other Vista Kartika Other XP Khmer UI Other 7 KodchiangUPC Thai 2000 Kokila Other 7 Lao UI Other 7 Latha Indic XP Leelawadee Thai Vista Leelawadee UI Thai 8.1 Levenim MT Hebrew 98 LilyUPC Thai 2000 Lucida Console Latin, Greek, Cyrillic 98 Lucida Sans Latin, Greek, Cyrillic 98 Lucida Sans Unicode Latin, Greek, Cyrillic 98 Malgun Gothic Korean Vista Mangal Indic XP Marlett Symbols 95 Meiryo Japanese Vista Meiryo UI Japanese 7 Microsoft Himalaya Other Vista Microsoft JhengHei Chinese Vista Microsoft JhengHei UI Chinese 8 Microsoft New Tai Lue Other 7 Microsoft PhagsPa Other 7 Microsoft Sans Serif Latin, Greek, Cyrillic 2000 Microsoft Tai Le Other 7 Microsoft Uighur Other Vista Microsoft YaHei Chinese Vista Microsoft YaHei UI Chinese 8 Microsoft Yi Baiti Other Vista MingLiU Chinese Vista MingLiU_HKSCS Chinese Vista MingLiU_HKSCS-ExtB Chinese Vista MingLiU-ExtB Chinese Vista Miriam Hebrew 2000 Miriam Fixed Hebrew 2000 Modern Latin 95 Mongolian Baiti Other Vista MoolBoran Other Vista MS Gothic Japanese 2000 MS Mincho Japanese 2000 MS PGothic Japanese Vista MS PMincho Japanese Vista MS Serif Latin 95 MS Sans Serif Latin 95 MS UI Gothic Japanese Vista MV Boli Other XP Myanmar Text Other 8 Narkisim Hebrew 2000 Neue Haas Grotesk Text Pro Latin 10 Nirmala UI Indic 8 NSimSun Chinese Vista Nyala African Vista Palatino Linotype Latin, Greek, Cyrillic 2000 Plantagenet Cherokee Other Vista PMingLiU Chinese 2000 PMingLiU-ExtB Chinese Vista Raavi Indic XP Rockwell Nova Latin, Greek, Cyrillic 10 Rod Hebrew 2000 Roman Latin 98 Sakkal Majalla Arabic 7 Sanskrit Text Other 10 Script Latin 98 Segoe MDL2 Assets Symbols 10 Segoe Print Latin, Greek, Cyrillic Vista Segoe Script Latin, Greek, Cyrillic Vista Segoe UI Latin, Greek, Cyrillic Vista Segoe UI Emoji Symbols 10 Segoe UI Historic Other 10 Segoe UI Light Latin, Greek, Cyrillic 7 Segoe UI Semibold Latin, Greek, Cyrillic 7 Segoe UI Semilight Latin, Greek, Cyrillic 8 Segoe UI Symbol Symbols 7 Shonar Bangla Indic 7 Shruti Indic XP SimHei Chinese 2000 Simplified Arabic Arabic 2000 Simplified Arabic Fixed Arabic 2000 SimSun Chinese 2000 SimSun-ExtB Chinese Vista Sitka Banner Latin, Greek, Cyrillic 8.1 Sitka Display Latin, Greek, Cyrillic 8.1 Sitka Heading Latin, Greek, Cyrillic 8.1 Sitka Small Latin, Greek, Cyrillic 8.1 Sitka Subheading Latin, Greek, Cyrillic 8.1 Sitka Text Latin, Greek, Cyrillic 8.1 Small Fonts Latin 95 Sylfaen Other XP Symbol Symbols 95 System Latin 95 Tahoma Latin, Greek, Cyrillic 95 Terminal Latin 95 Times New Roman Latin, Greek, Cyrillic 95 Traditional Arabic Arabic 2000 Trebuchet MS Latin, Greek, Cyrillic 2000 Tunga Indic XP UD Digi Kyokasho N-B Japanese 10 UD Digi Kyokasho NK-B Japanese 10 UD Digi Kyokasho NK-R Japanese 10 UD Digi Kyokasho NP-B Japanese 10 UD Digi Kyokasho NP-R Japanese 10 UD Digi Kyokasho N-R Japanese 10 Urdu Typesetting Arabic 8 Utsaah Indic 7 Vani Indic 7 Verdana Latin, Greek, Cyrillic 95 Verdana Pro Latin, Greek, Cyrillic 10 Vijaya Indic 7 Vrinda Indic XP Webdings Symbols 98 Wingdings Symbols 95 Yu Gothic Japanese 8.1 Yu Gothic UI Japanese 8.1 Yu Mincho Japanese 8.1 Function Objects - Definition & Usage | AutoHotkey v2 Function Objects "Function object" usually means any of the following: A Func or Closure object, which represents an actual function; either built-in or defined by the script. A user-defined object which can be called like a function. This is sometimes also referred to as a "functor". Any other object which can be called like a function, such as a BoundFunc object or a JavaScript function object returned by a COM method. Function objects can be used with the following: CallbackCreate Gui events For ... in HotIf Hotkey Hotstring InputHook (OnEnd, OnChar, OnKeyDown, OnKeyUp) Menu.Add OnClipboardChange OnError OnExit OnMessage RegEx callouts SetTimer Sort To determine whether an object appears to be callable, use one of the following: Value.HasMethod() works with all AutoHotkey values and objects by default, but allows HasMethod to be overridden for some objects or classes. For COM objects, this will typically fail (throw an exception or produce the wrong result) unless the COM object is actually an AutoHotkey object from another process. HasMethod(Value) works with all AutoHotkey values and objects and cannot be overridden, but will return false if the presence of a Call method cannot be determined. An exception is thrown if Value is a ComObject. User-Defined User-defined function objects must define a Call method containing the implementation of the "function". class YourClassName { Call(a, b) { ; Declare parameters as needed, or an array*. ;... return c } ;... } This applies to instances of YourClassName, such as the object returned by YourClassName(). Replacing Call with static Call would instead override what occurs when YourClassName itself is called. Examples The following example defines a function array which can be called; when called, it calls each element of the array in turn. class FuncArrayType extends Array { Call(params*) { ; Call a list of functions. for fn in this fn(params*) } } ; Create an array of functions. funcArray := FuncArrayType() ; Add some functions to the array (can be done at any point). funcArray.Push(One) funcArray.Push(Two) ; Create an object which uses the array as a method. obj := {method: funcArray} ; Call the method (and consequently both One and Two). obj.method("2nd") ; Call it as a function. (obj.method)("1st", "2nd") One(param1, param2) { ListVars MsgBox } Two(param1, param2) { ListVars MsgBox } BoundFunc Object Acts like a function, but just passes predefined parameters to another function. There are two ways that BoundFunc objects can be created: By calling the Func.Bind method, which binds parameter values to a function. By calling the ObjBindMethod function, which binds parameter values and a method name to a target object. BoundFunc objects can be called as shown in the example below. When the BoundFunc is called, it calls the function or method to which it is bound, passing a combination of bound parameters and the caller's parameters. Unbound parameter positions are assigned positions from the caller's parameter list, left to right. For example: fn := RealFn.Bind(1) ; Bind first parameter only fn(2) ; Shows "1, 2" fn.Call(3) ; Shows "1, 3" fn := RealFn.Bind(, 1) ; Bind second parameter only fn(2) ; Shows "2, 1" fn.Call(3) ; Shows "3, 1" fn(, 4) ; Error: 'a' was omitted RealFn(a, b, c:="c") { MsgBox a ", " b } ObjBindMethod can be used to bind to a method even when it isn't possible to retrieve a reference to the method itself. For example: Shell := ComObject("Shell.Application") RunBox := ObjBindMethod(Shell, "FileRun") ; Show the Run dialog. RunBox For a more complex example, see SetTimer. Other properties and methods are inherited from Func, but do not reflect the properties of the target function or method (which is not required to be implemented as a function). The BoundFunc acts as an anonymous variadic function with no other formal parameters, similar to the fat arrow function below: Func_Bind(fn, bound_args*) { return (args*) => (args.InsertAt(1, bound_args*), fn(args*)) } Image Handles - Syntax & Usage | AutoHotkey v2 Image Handles To use an icon or bitmap handle in place of an image filename, use the following syntax: HBITMAP:bitmap-handle HICON:icon-handle Replace bitmap-handle or icon-handle with the actual handle value. For example, "hicon:" handle, where handle is a variable containing an icon handle. The following things support this syntax: Gui.AddPicture (and GuiCtrl.Value when setting a Picture control's content). IL_Add LoadPicture

SB.SetIcon ImageSearch TraySetIcon or Menu.SetIcon A bitmap or icon handle is a numeric value which identifies a bitmap or icon in memory. The majority of scripts never need to deal with handles, as in most cases AutoHotkey takes care of loading the image from file and freeing it when it is no longer needed. The syntax shown above is intended for use when the script obtains an icon or bitmap handle from another source, such as by sending the WM_GETICON message to a window. It can also be used in combination with LoadPicture to avoid loading an image from file multiple times. By default, AutoHotkey treats the handle as though it loaded the image from file - for example, a bitmap used on a Picture control is deleted when the GUI is destroyed, and an image will generally be deleted immediately if it needs to be resized. To avoid this, put an asterisk between the colon and handle. For example: "hbitmap:*" handle. With the exception of ImageSearch, this forces the function to take a copy of the image. Examples Shows a menu of the first n files matching a pattern, and their icons. pattern := A_ScriptDir "*" n := 15 ; Create a menu. Fmenu := Menu() ; Allocate memory for a SHFILEINFOW struct. fileinfo := Buffer(fisize := A_PtrSize + 688) Loop Files, pattern, "FD" { ; Add a menu item for each file. Fmenu.Add A_LoopFileName, (*) => "" ; Do nothing. ; Get the file's icon. if DllCall("shell32 \SHGetFileInfoW", "WStr", A_LoopFileFullPath , "UInt", 0, "Ptr", fileinfo, "UInt", fisize, "UInt", 0x100) { hicon := NumGet(fileinfo, 0, "Ptr") ; Set the menu item's icon. Fmenu.SetIcon A_Index "&", "HICON:" hicon ; Because we used ":" and not ":*", the icon will be automatically ; freed when the program exits or if the menu or item is deleted. } } until A_Index = n Fmenu.Show See also: LoadPicture. Labels - Syntax & Usage | AutoHotkey v2 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 Language Codes | AutoHotkey v2 Language Codes The following list contains the language name that corresponds to each language code that can be contained in the A_Language variable. The language code itself is the last four digits on the left side of the equal sign below. For example, if A_Language contains 0436, the system's default language is Afrikaans. Note: Codes that contain letters might use either uppercase or lowercase. You can compare A_Language directly to one or more of the 4-digit codes below; for example: if (A_Language = "0436"). Alternatively, you can paste the entire list into a script and then access the name of the current language as demonstrated at the bottom of the list. languageCode_0436 := "Afrikaans" languageCode_041c := "Albanian" languageCode_0401 := "Arabic_Saudi_Arabia" languageCode_0801 := "Arabic_Iraq" languageCode_0c01 := "Arabic_Egypt" languageCode_1001 := "Arabic_Libya" languageCode_1401 := "Arabic_Algeria" languageCode_1801 := "Arabic_Morocco" languageCode_1c01 := "Arabic_Tunisia" languageCode_2001 := "Arabic_Oman" languageCode_2401 := "Arabic_Yemen" languageCode_2801 := "Arabic_Syria" languageCode_2c01 := "Arabic_Jordan" languageCode_3001 := "Arabic_Lebanon" languageCode_3401 := "Arabic_Kuwait" languageCode_3801 := "Arabic_UAE" languageCode_3c01 := "Arabic_Bahrain" languageCode_4001 := "Arabic_Qatar" languageCode_042b := "Armenian" languageCode_042c := "Azeri_Latin" languageCode_082c := "Azeri_Cyrillic" languageCode_042d := "Basque" languageCode_0423 := "Belarusian" languageCode_0402 := "Bulgarian" languageCode_0403 := "Catalan" languageCode_0404 := "Chinese_Taiwan" languageCode_0804 := "Chinese_PRC" languageCode_0c04 := "Chinese_Hong_Kong" languageCode_1004 := "Chinese_Singapore" languageCode_1404 := "Chinese_Macau" languageCode_041a := "Croatian" languageCode_0405 := "Czech" languageCode_0406 := "Danish" languageCode_0413 := "Dutch_Standard" languageCode_0813 := "Dutch_Belgian" languageCode_0409 := "English_United_States" languageCode_0809 := "English_United_Kingdom" languageCode_0c09 := "English_Australian" languageCode_1009 := "English_Canadian" languageCode_1409 := "English_New_Zealand" languageCode_1809 := "English_Irish" languageCode_1c09 := "English_South_Africa" languageCode_2009 := "English_Jamaica" languageCode_2409 := "English_Caribbean" languageCode_2809 := "English_Belize" languageCode_2c09 := "English_Trinidad" languageCode_3009 := "English_Zimbabwe" languageCode_3409 := "English_Philippines" languageCode_0425 := "Estonian" languageCode_0438 := "Faeroese" languageCode_0429 := "Farsi" languageCode_040b := "Finnish" languageCode_040c := "French_Standard" languageCode_080c := "French_Belgian" languageCode_0c0c := "French_Canadian" languageCode_100c := "French_Swiss" languageCode_140c := "French_Luxembourg" languageCode_180c := "French_Monaco" languageCode_0437 := "Georgian" languageCode_0407 := "German_Standard" languageCode_0807 := "German_Swiss" languageCode_0c07 := "German_Austrian" languageCode_1007 := "German_Luxembourg" languageCode_1407 := "German_Liechtenstein" languageCode_0408 := "Greek" languageCode_040d := "Hebrew" languageCode_0439 := "Hindi" languageCode_040e := "Hungarian" languageCode_040f := "Icelandic" languageCode_0421 := "Indonesian" languageCode_0410 := "Italian_Standard" languageCode_0810 := "Italian_Swiss" languageCode_0411 := "Japanese" languageCode_043f := "Kazakh" languageCode_0457 := "Konkani" languageCode_0412 := "Korean" languageCode_0426 := "Latvian" languageCode_0427 := "Lithuanian" languageCode_042f := "Macedonian" languageCode_043e := "Malay_Malaysia" languageCode_083e := "Malay_Brunei_Darussalam" languageCode_044e := "Marathi" languageCode_0414 := "Norwegian_Bokmal" languageCode_0814 := "Norwegian_Nynorsk" languageCode_0415 := "Polish" languageCode_0416 := "Portuguese_Brazilian" languageCode_0816 := "Portuguese_Standard" languageCode_0418 := "Romanian" languageCode_0419 := "Russian" languageCode_044f := "Sanskrit" languageCode_081a := "Serbian_Latin" languageCode_0c1a := "Serbian_Cyrillic" languageCode_041b := "Slovak" languageCode_0424 := "Slovenian" languageCode_040a := "Spanish_Traditional_Sort" languageCode_080a := "Spanish_Mexican" languageCode_0c0a := "Spanish_Modern_Sort" languageCode_100a := "Spanish_Guatemala" languageCode_140a := "Spanish_Costa_Rica" languageCode_180a := "Spanish_Panama" languageCode_1c0a := "Spanish_Dominican_Republic" languageCode_200a := "Spanish_Venezuela" languageCode_240a := "Spanish_Colombia" languageCode_280a := "Spanish_Peru" languageCode_2c0a := "Spanish_Argentina" languageCode_300a := "Spanish_Ecuador" languageCode_340a := "Spanish_Chile" languageCode_380a := "Spanish_Uruguay" languageCode_3c0a := "Spanish_Paraguay" languageCode_400a := "Spanish_Bolivia" languageCode_440a := "Spanish_El_Salvador" languageCode_480a := "Spanish_Honduras" languageCode_4c0a := "Spanish_Nicaragua" languageCode_500a := "Spanish_Puerto_Rico" languageCode_0441 := "Swahili" languageCode_041d := "Swedish" languageCode_081d := "Swedish_Finland" languageCode_0449 := "Tamil" languageCode_0444 := "Tatar" languageCode_041e := "Thai" languageCode_041f := "Turkish" languageCode_0422 := "Ukrainian" languageCode_0420 := "Urdu" languageCode_0443 := "Uzbek_Latin" languageCode_0843 := "Uzbek_Cyrillic" languageCode_042a := "Vietnamese" the_language := languageCode_%A_Language% ; Get the name of the system's default language. MsgBox the_language ; Display the language name. Long Paths | AutoHotkey v2 Long Paths In general, programs are affected by two kinds of path length limitations: Functions provided by the operating system generally limit paths to 259 characters, with some exceptions. Code for dealing with paths within the program may rely on the first limitation to simplify the code, effectively placing another 259 character limitation. These limitations are often referred to as "MAX_PATH limitations", after the constant MAX_PATH, which has the value 260. One character is generally reserved for a null terminator, leaving 259 characters for the actual path. AutoHotkey removes the second kind in most cases, which enables the script to work around the first kind. There are two ways to do this: Long paths can be enabled for AutoHotkey and all other long-path-aware programs on Windows 10. Refer to the Microsoft documentation for details. In short, this enables most functions to transparently work with long paths, but requires Windows 10 version 1607 or later. In many cases, prefixing the path with \\?\ enables it to exceed the usual limit. However, some system functions do not support it (or long paths in general). See Known Limitations for details. Long Path Prefix If supported by the underlying system function, the \\?\ prefix -- for example, in \\?\C:\My Folder -- increases the limit to 32,767 characters. However, it does this by skipping path normalization. Some elements of the path which would normally be removed or altered by normalization instead become part of the file's actual path. Care must be taken as this can allow the creation of paths that "normal" programs cannot access. In particular, normalization: Resolves relative paths such as dir\file.ext, \file.ext and C:file.ext (note the absence of a slash). Resolves relative components such as \.. and \.. Canonicalizes component/directory separators, replacing / with \ and eliminating redundant separators. Trims certain characters, such as a single period at the end of a component (dir.\file) or trailing spaces and periods (dir\filename . .). A path can be normalized explicitly by passing it to GetFullPathName via the function defined below, before applying the prefix. For example: MsgBox "\\?\" NormalizePath("..\file.ext") NormalizePath (path) { cc := DllCall("GetFullPathName", "str", path, "uint", 0, "ptr", 0, "ptr", 0, "uint") buf := Buffer(cc*2) DllCall("GetFullPathName", "str", path, "uint", cc, "ptr", buf, "ptr", 0) return StrGet(buf) } A path with the \\?\ prefix can also be normalized by this function. However, in that case the working directory is never used, and the root is \\?\ (for example, \\?\C:\.. resolves to \\?\ whereas C:\.. resolves to C:\). Known Limitations Even when the path itself is not limited to 259 characters, each component (file or directory name) cannot exceed the hard limit imposed by the file system (usually 255 characters). These do not support long paths due to limitations of the underlying system function(s): DllCall (for DllFile and Function) DirCopy DirDelete, unless Recurse is false DirMove, unless the R option is used FileCreateShortcut FileGetShortcut FileRecycle SoundPlay (for this, the limit is 127 characters) DriveSetLabel and DriveGet variants (except DriveGetType) Built-in variables which return special folder paths (for which long paths might be impossible anyway): A_AppData, A_Desktop, A_MyDocuments, A_ProgramFiles, A_Programs, A_StartMenu, A_Startup and Common variants, A_Temp and A_WinDir SetWorkingDir and A_WorkingDir support long paths

only when Windows 10 long path awareness is enabled, since the \\?\ prefix cannot be used. If the working directory exceeds MAX_PATH, it becomes impossible to launch programs with Run. These limitations are imposed by the OS. It does not appear to be possible to run an executable with a full path which exceeds MAX_PATH. That being the case, it would not be possible to fully test any changes aimed at supporting longer executable paths. Therefore, MAX_PATH limits have been left in place for the following: ahk_exe The default script's path, which is based on the current executable's path. Retrieval of the AutoHotkey installation directory, which is used by A_AhkPath in compiled scripts and may be used to launch Window Spy or the help file. WinGetProcessPath. WinGetProcessName (this theoretically isn't a problem since it is applied only to the name portion, and NTFS only supports names up to 255 chars). Long #Include paths shown in error messages may be truncated arbitrarily. Creating a Keyboard Macro or Mouse Macro | AutoHotkey v2 Creating a Keyboard Macro or Mouse Macro A macro is a series of scripted actions that is "played" upon demand. The most common activity of a macro is to send simulated keystrokes and mouse clicks to one or more windows. Such windows respond to each keystroke and mouse click as though you had performed it manually, which allows repetitive tasks to be automated with high speed and reliability. One of the most convenient ways to play back a macro is to assign it to a hotkey or hotstring. For example, the following hotkey would create an empty e-mail message and prepare it for a certain type recipient, allowing you to personalize it prior to sending: ^!s:: ; Control+Alt+S hotkey. { if not WinExist("Inbox - Microsoft Outlook") return ; Outlook isn't open to the right section, so do nothing. WinActivate ; Activate the window found by the above function. Send "^n" ; Create new/blank e-mail via Control+N. WinWaitActive "Untitled Message" Send "{Tab 2}Product Recall for ACME Rocket Skates" ; Set the subject line. Send "{Tab}Dear Sir or Madam,{Enter 2}We have recently discovered a minor defect ..." ; etc. } ; This brace marks the end of the hotkey. Hotkey macros like the above are especially useful for tasks you perform several times per day. By contrast, macros used less often can each be kept in a separate script accessible by means of a shortcut in the Start Menu or on the desktop. To start creating your own macros and hotkeys right away, please read the Quick-start Tutorial. Overriding or Disabling Hotkeys | AutoHotkey v2 Overriding or Disabling Hotkeys You can disable all built-in Windows hotkeys except Win+L and Win+U by making the following change to the registry (this should work on all OSes but a reboot is probably required): HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer NoWinKeys REG_DWORD 0x00000001 (1) But read on if you want to do more than just disable them all. Hotkeys owned by another application can be overridden or disabled simply by assigning them to an action in the script. The most common use for this feature is to change the hotkeys that are built into Windows itself. For example, if you wish Win+E (the shortcut key that launches Windows Explorer) to perform some other action, use this: #e::MsgBox "This hotkey is now owned by the script." In the next example, the Win+R hotkey, which is used to open the RUN window, is completely disabled: #r::return Similarly, to disable both Win, use this: LWin::return RWin::return To disable or change an application's non-global hotkey (that is, a shortcut key that only works when that application is the active window), consider the following example which disables Ctrl+P (Print) only for Notepad, leaving the key in effect for all other types of windows: $^p:: { if WinActive("ahk_class Notepad") return ; i.e. do nothing, which causes Control-P to do nothing in Notepad. Send "^p" } In the above example, the $ prefix is needed so that the hotkey can "send itself" without activating itself (which would otherwise trigger a warning dialog about an infinite loop). See also: context-sensitive hotkeys. You can try out any of the above examples by copying them into a new text file such as "Override.ahk", then launching the file. Script Performance | AutoHotkey v2 Script Performance The following functions may affect performance depending on the nature of the script: SendMode, SetKeyDelay, SetMouseDelay, SetWinDelay, SetControlDelay, and SetDefaultMouseSpeed. Built-in Performance Features Each script is semi-compiled while it is being loaded and syntax-checked. In addition to detecting some errors early, this also greatly improves runtime performance. Here are some of the technical details of the optimization process (semi-compiling): Loops, blocks, IFs, ELSEs and other control flow statements are given the memory addresses of their related jump-points in the script. Each statement name is replaced by an address in a jump table. Each expression is tokenized and converted from infix to postfix. Each reference to a variable or function is resolved to a memory address, unless it is dynamic. Literal integers in expressions are replaced with binary integers. The destination of each Goto is resolved to a memory address unless it is a variable. In addition, during script execution, binary numbers are cached in variables to avoid conversions to/from strings. Regular Expressions (RegEx) - Quick Reference | AutoHotkey v2 Regular Expressions (RegEx) - Quick Reference Table of Contents Fundamentals Options (case sensitive) Commonly Used Symbols and Syntax Fundamentals Match anywhere: By default, a regular expression matches a substring anywhere inside the string to be searched. For example, the regular expression abc matches abc123, 123abc, and 123abcxyz. To require the match to occur only at the beginning or end, use an anchor. Escaped characters: Most characters like abc123 can be used literally inside a regular expression. However, the characters \.*?+[{|()^$ must be preceded by a backslash to be seen as literal. For example, \. is a literal period and \\ is a literal backslash. Escaping can be avoided by using \Q...\E. For example: \QLiteral Text\E. Case-sensitive: By default, regular expressions are case-sensitive. This can be changed via the "i" option. For example, the pattern i)abc searches for "abc" without regard to case. See below for other modifiers. Options (case sensitive) At the very beginning of a regular expression, specify zero or more of the following options followed by a close-parenthesis. For example, the pattern im)abc would search for "abc" with the case-insensitive and multiline options (the parenthesis may be omitted when there are no options). Although this syntax breaks from tradition, it requires no special delimiters (such as forward-slash), and thus there is no need to escape such delimiters inside the pattern. In addition, performance is improved because the options are easier to parse. Option Description i Case-insensitive matching, which treats the letters A through Z as identical to their lowercase counterparts. m Multiline. Views Haystack as a collection of individual lines (if it contains newlines) rather than as a single continuous line. Specifically, it changes the following: 1) Circumflex (^) matches immediately after all internal newlines -- as well as at the start of Haystack where it always matches (but it does not match after a newline at the very end of Haystack). 2) Dollar-sign ($) matches before any newlines in Haystack (as well as at the very end where it always matches). For example, the pattern m)^abc$ matches xyz`r`nabc. But without the "m" option, it wouldn't match. The "D" option is ignored when "m" is present. s DotAll. This causes a period (.) to match all characters including newlines (normally, it does not match newlines). However, two dots are required to match a CRLF newline sequence (`r`n), not one. Regardless of this option, a negative class such as [^a] always matches newlines. x Ignores whitespace characters in the pattern except when escaped or inside a character class. The characters `n and `t are among those ignored because by the time they get to PCRE, they are already raw/literal whitespace characters (by contrast, \n and \t are not ignored because they are PCRE escape sequences). The "x" option also ignores characters between a non-escaped # outside a character class and the next newline character, inclusive. This makes it possible to include comments inside complicated patterns. However, this applies only to data characters; whitespace may never appear within special character sequences such as (?(, which begins a conditional subpattern. A Forces the pattern to be anchored; that is, it can match only at the start of Haystack. Under most conditions, this is equivalent to explicitly anchoring the pattern by means such as "^". D Forces dollar-sign ($) to match at the very end of Haystack, even if Haystack's last item is a newline. Without this option, $ instead matches right before the final newline (if there is one). Note: This option is ignored when the "m" option is present. J Allows duplicate named subpatterns. This can be useful for patterns in which only one of a collection of identically-named subpatterns can match. Note: If more than one instance of a particular name matches something, only the leftmost one is stored. Also, variable names are not case-sensitive. U Ungreedy. Makes the quantifiers *, ?, + and {min,max} consume only those characters absolutely necessary to form a match, leaving the remaining ones available for the next part of the pattern. When the "U" option is not in effect, an individual quantifier can be made non-greedy by following it with a question mark. Conversely, when "U" is in effect, the question mark makes an individual quantifier greedy. X PCRE_EXTRA. Enables PCRE features that are incompatible with Perl. Currently, the only such feature is that any backslash in a pattern that is followed by a letter that has no special meaning causes an exception to be thrown. This option helps reserve unused backslash sequences for future use. Without this option, a backslash followed by a letter with no special meaning is treated as a literal (e.g. \g and g are both recognized as a literal g). Regardless of this option, non-alphabetic backslash sequences that have no special meaning are always treated as literals (e.g. \/ and / are both recognized as forward-slash). S Studies the pattern to try improve its performance. This is useful when a particular pattern (especially a complex one) will be executed many times. If PCRE finds a way to improve performance, that discovery is stored alongside the pattern in the cache for use by subsequent executions of the same pattern (subsequent uses of that pattern should also specify the S option because finding a match in the cache requires that the option letters exactly match, including their order). C Enables the auto-callout mode. See Regular Expression Callouts for more info. `a Enables recognition of additional newline markers. By default, only `r`n, `n and `r are recognized. With this option enabled, `v/VT/vertical tab/chr(0xB), `f/FF/formfeed/chr(0xC), NEL/next-line/chr(0x85), LS/line separator/chr(0x2028) and PS/paragraph separator/chr(0x2029) are also recognized. The `a, `n and `r options affect the behavior of anchors (^ and $) and the dot/period pattern. `a also puts (*BSR_UNICODE) into effect, which causes \R to match any kind of newline. By default, \R matches `n, `r and `r`n; this behaviour can be restored by combining options as follows: `a)(*BSR_ANYCRLF) `n Causes a solitary linefeed (`n) to be the only recognized newline marker (see above). `r Causes a solitary carriage return (`r) to be the only recognized newline marker (see above). Note: Spaces and tabs may optionally be used to separate each option from the next. Commonly Used Symbols and Syntax Element Description . By default, a dot matches any single character except `r (carriage return) or `n (linefeed), but this can be changed by using the DotAll (s), linefeed (`n), carriage return (`r), or `a options. For example, ab. matches abc and abz and ab_. * An asterisk matches zero or more of the preceding character, class, or subpattern. For example, a* matches ab and aaab. It also matches at the very beginning of any string that contains no "a" at all. Wildcard: The dot-star pattern .* is one of the most permissive because it matches zero or more occurrences of any character (except newline: `r and `n). For example, abc.*123 matches abcAnything123 as well as abc123. ? A question mark matches zero or one of the preceding character, class, or subpattern. Think of this as "the preceding item is optional". For example, colou?r matches both color and colour because the "u" is optional. + A plus sign matches one or more of the preceding character, class, or subpattern. For example a+ matches ab and aaab. But unlike a* and a?, the pattern a+ does not match at the beginning of strings that lack an "a" character. {min,max} Matches between min and max occurrences of the preceding character, class, or subpattern. For example, a{1,2} matches ab but only the first two a's in aaab. Also, {3} means exactly 3 occurrences, and {3,} means 3 or more occurrences. Note: The specified numbers must be less than 65536, and the first must be less than or equal to the second. [...] Classes of characters: The square brackets enclose a list or range of characters (or both). For example,

[abc] means "any single character that is either a, b or c". Using a dash in between creates a range; for example, [a-z] means "any single character that is between lowercase a and z (inclusive)". Lists and ranges may be combined; for example [a-zA-Z0-9_] means "any single character that is alphanumeric or underscore". A character class may be followed by *, ?, +, or {min,max}. For example, [0-9]+ matches one or more occurrence of any digit; thus it matches xyz123 but not abcxyz. The following POSIX named sets are also supported via the form [[:xxx:]], where xxx is one of the following words: alnum, alpha, ascii (0-127), blank (space or tab), cntrl (control character), digit (0-9), xdigit (hex digit), print, graph (print excluding space), punct, lower, upper, space (whitespace), word (same as \w). Within a character class, characters do not need to be escaped except when they have special meaning inside a class; e.g. [\^a], [a\-b], [a\]], and [\\a]. [^...] Matches any single character that is not in the class. For example, [^/]* matches zero or more occurrences of any character that is not a forward-slash, such as http://. Similarly, [^0-9xyz] matches any single character that isn't a digit and isn't the letter x, y, or z. \d Matches any single digit (equivalent to the class [0-9]). Conversely, capital \D means "any non-digit". This and the other two below can also be used inside a class; for example, [\d.-] means "any single digit, period, or minus sign". \s Matches any single whitespace character, mainly space, tab, and newline (`r and `n). Conversely, capital \S means "any non-whitespace character". \w Matches any single "word" character, namely alphanumeric or underscore. This is equivalent to [a-zA-Z0-9_]. Conversely, capital \W means "any non-word character". ^$ Circumflex (^) and dollar sign ($) are called anchors because they don't consume any characters; instead, they tie the pattern to the beginning or end of the string being searched. ^ may appear at the beginning of a pattern to require the match to occur at the very beginning of a line. For example, ^abc matches abc123 but not 123abc. $ may appear at the end of a pattern to require the match to occur at the very end of a line. For example, abc$ matches 123abc but not abc123. The two anchors may be combined. For example, ^abc$ matches only abc (i.e. there must be no other characters before or after it). If the text being searched contains multiple lines, the anchors can be made to apply to each line rather than the text as a whole by means of the "m" option. For example, m)^abc$ matches 123`r`nabc`r`n789. But without the "m" option, it wouldn't match. \b \b means "word boundary", which is like an anchor because it doesn't consume any characters. It requires the current character's status as a word character (\w) to be the opposite of the previous character's. It is typically used to avoid accidentally matching a word that appears inside some other word. For example, \bcat\b doesn't match catfish, but it matches cat regardless of what punctuation and whitespace surrounds it. Capital \B is the opposite: it requires that the current character not be at a word boundary. | The vertical bar separates two or more alternatives. A match occurs if any of the alternatives is satisfied. For example, gray|grey matches both gray and grey. Similarly, the pattern gr(a|e)y does the same thing with the help of the parentheses described below. (...) Items enclosed in parentheses are most commonly used to: Determine the order of evaluation. For example, (Sun|Mon|Tues|Wednes|Thurs|Fri|Satur)day matches the name of any day. Apply *, ?, +, or {min,max} to a series of characters rather than just one. For example, (abc)+ matches one or more occurrences of the string "abc"; thus it matches abcabc123 but not ab123 or bc123. Capture a subpattern such as the dot-star in abc(.*)xyz. For example, RegExMatch stores the substring that matches each subpattern in its output array. Similarly, RegExReplace allows the substring that matches each subpattern to be reinserted into the result via backreferences like $1. To use the parentheses without the side-effect of capturing a subpattern, specify ?: as the first two characters inside the parentheses; for example: (?:.*) Change options on-the-fly. For example, (?im) turns on the case-insensitive and multiline options for the remainder of the pattern (or subpattern if it occurs inside a subpattern). Conversely, (?-im) would turn them both off. All options are supported except DPS`r`n`a. \t\retc. These escape sequences stand for special characters. The most common ones are \t (tab), \r (carriage return), and \n (linefeed). In AutoHotkey, an accent (`) may optionally be used in place of the backslash in these cases. Escape sequences in the form \xhh are also supported, in which hh is the hex code of any ANSI character between 00 and FF. \R matches `r`n, `n and `r (however, \R inside a character class is merely the letter "R"). \p{xx}\P{xx}\X Unicode character properties. \p{xx} matches a character with the xx property while \P{xx} matches any character without the xx property. For example, \pL matches any letter and \p{Lu} matches any upper-case letter. \X matches any number of characters that form an extended Unicode sequence. For a full list of supported property names and other details, search for "\p{xx}" at www.pcre.org/pcre.txt. (*UCP) For performance, \d, \D, \s, \S, \w, \W, \b and \B recognize only ASCII characters by default. If the pattern begins with (*UCP), Unicode properties will be used to determine which characters match. For example, \w becomes equivalent to [\p{L}\p{N}_] and \d becomes equivalent to \p{Nd}. Greed: By default, *, ?, +, and {min,max} are greedy because they consume all characters up through the last possible one that still satisfies the entire pattern. To instead have them stop at the first possible character, follow them with a question mark. For example, the pattern <.+> (which lacks a question mark) means: "search for a <, followed by one or more of any character, followed by a >". To stop this pattern from matching the entire string *text*, append a question mark to the plus sign: <.+?>. This causes the match to stop at the first '>' and thus it matches only the first tag . *Look-ahead and look-behind assertions: The groups (?=...), (?!...), (?<=...), and (?= 2) mark := StrGet(NumGet(A_EventInfo, 36 + pad + A_PtrSize*3, "Int"), "UTF-8") For more information, see pcre.txt, NumGet and A_PtrSize. Auto-Callout Including C in the options of the pattern enables the auto-callout mode. In this mode, RegEx callouts equivalent to (?C255) are inserted before each item in the pattern. For example, the following template may be used to debug regular expressions: ; Call RegExMatch with auto-callout option C. RegExMatch("xxxabc123xyz", "C)abc.*xyz") ; Define the default RegEx callout function. pcre_callout(Match, CalloutNumber, FoundPos, Haystack, NeedleRegEx) { ; See pcre.txt for descriptions of these fields. start_match := NumGet(A_EventInfo, 12 + A_PtrSize*2, "Int") current_position := NumGet (A_EventInfo, 16 + A_PtrSize*2, "Int") pad := A_PtrSize=8 ? 4 : 0 pattern_position := NumGet(A_EventInfo, 28 + pad + A_PtrSize*3, "Int") next_item_length := NumGet(A_EventInfo, 32 + pad + A_PtrSize*3, "Int") ; Point out >>current match<<. _HAYSTACK:=SubStr(Haystack, 1, start_match) . ">>" SubStr(Haystack, start_match + 1, current_position - start_match) . "<<" SubStr(Haystack, current_position + 1) ; Point out >>next item to be evaluated<<. _NEEDLE:= SubStr (NeedleRegEx, 1, pattern_position) . ">>" SubStr(NeedleRegEx, pattern_position + 1, next_item_length) . "<<" SubStr(NeedleRegEx, pattern_position + 1 + next_item_length) ListVars ; Press Pause to continue. Pause } Remarks RegEx callouts are executed on the current quasi-thread, but the previous value of A_EventInfo will be restored after the RegEx callout function returns. PCRE is optimized to abort early in some cases if it can determine that a match is not possible. For all RegEx callouts to be called in such cases, it may be necessary to disable these optimizations by specifying (*NO_START_OPT) at the start of the pattern. Remapping Keys (Keyboard, Mouse and Joystick) | AutoHotkey v2 Remapping Keys (Keyboard, Mouse and Joystick) Table of Contents Introduction Remapping the Keyboard and Mouse 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 Remapping a Joystick to Keyboard or Mouse | AutoHotkey v2 Remapping a Joystick to Keyboard or Mouse Table of Contents Important Notes Making a Joystick Button Send Keystrokes or Mouse Clicks Different Approaches Auto-repeating a Keystroke Context-sensitive Joystick Buttons Using a Joystick as a Mouse Making Other Joystick Controls Send Keystrokes or Mouse Clicks Joystick Axes Joystick POV Hat Auto-repeating a Keystroke Remarks Related Topics Important Notes Although a joystick button or axis can be remapped to become a key or mouse button, it cannot be remapped to some other joystick button or axis. That would be possible only with the help of a joystick emulator such as vJoy. AutoHotkey identifies each button on a joystick with a unique number between 1 and 32. To determine these numbers, use the joystick test script. Making a Joystick Button Send Keystrokes or Mouse Clicks Different Approaches Below are three approaches, starting at the simplest and ending with the most complex. The most complex method works in the broadest variety of circumstances (such as games that require a key or mouse button to be held down). Method #1 This method sends simple keystrokes and mouse clicks. For example: Joy1::Send "{Left}" ; Have button #1 send a left-arrow keystroke. Joy2::Click ; Have button #2 send a click of left mouse button. Joy3::Send "a{Esc}{Space}{Enter}" ; Have button #3 send the letter "a" followed by Escape, Space, and Enter. Joy4::Send "Sincerely,{Enter}John Smith" ; Have button #4 send a two-line signature. To have a button perform more than one function, put the first function beneath the button name and make the last line a return. For example: Joy5:: { Run "notepad" WinWait "Untitled - Notepad" WinActivate Send "This is the text that will appear in Notepad.{Enter}" } See the Key List for the complete list of keys and mouse/joystick buttons. Method #2 This method is necessary in cases where a key or mouse button must be held down for the entire time that you're holding down a joystick button. The following example makes the joystick's second button become the left-arrow key: Joy2:: { Send "{Left down}" ; Hold down the left-arrow key. KeyWait "Joy2" ; Wait for the user to release the joystick button. Send "{Left up}" ; Release the left-arrow key. } Method #3 This method is necessary in cases where you have more than one joystick hotkey of the type described in Method #2, and you sometimes press and release such hotkeys simultaneously. The following example makes the joystick's third button become the left mouse button: Joy3:: { Send "{LButton down}" ; Hold down the left mouse button. SetTimer WaitForButtonUp3, 10 } WaitForButtonUp3() { if GetKeyState("Joy3") ; The button is still, down, so keep waiting. return ; Otherwise, the button has been released. Send "{LButton up}" ; Release the left mouse button. SetTimer , 0 } Auto-repeating a Keystroke Some programs or games might require a key to be sent repeatedly (as though you are holding it down on the keyboard). The following example achieves this by sending spacebar keystrokes repeatedly while you hold down the joystick's second button: Joy2:: { Send "{Space down}" ; Press the spacebar down. SetTimer WaitForJoy2, 30 ; Reduce the number 30 to 20 or 10 to send keys faster. Increase it to send slower. } WaitForJoy2() { if not GetKeyState("Joy2") ; The button has been released. { Send "{Space up}" ; Release the spacebar. SetTimer , 0 ; Stop monitoring the button. return } ; Since above didn't "return", the button is still being held down. Send "{Space down}" ; Send another Spacebar keystroke. } Context-sensitive Joystick Buttons The #HotIf directive can be used to make selected joystick buttons perform a different action (or none at all) depending on any condition, such as the type of window that is active. Using a Joystick as a Mouse The Joystick-To-Mouse script converts a joystick into a mouse by remapping its buttons and axis control. Making Other Joystick Controls Send Keystrokes or Mouse Clicks To have a script respond to movement of a joystick's axis or POV hat, use SetTimer and GetKeyState. Joystick Axes The following example makes the joystick's X and Y axes behave like the arrow key cluster on a keyboard (left, right, up, and down): SetTimer WatchAxis, 5 WatchAxis() { static KeyToHoldDown := "" JoyX := GetKeyState("JoyX") ; Get position of X axis. JoyY := GetKeyState ("JoyY") ; Get position of Y axis. KeyToHoldDownPrev := KeyToHoldDown ; Prev now holds the key that was down before (if any). if JoyX > 70 KeyToHoldDown := "Right" else if JoyX < 30 KeyToHoldDown := "Left" else if JoyY > 70 KeyToHoldDown := "Down" else if JoyY < 30 KeyToHoldDown := "Up" else KeyToHoldDown := "" if KeyToHoldDown = KeyToHoldDownPrev ; The correct key is already down (or no key is needed). return ; Do nothing. ; Otherwise, release the previous key and press down the new key: SetKeyDelay -1 ; Avoid delays between keystrokes. if KeyToHoldDownPrev ; There is a previous key to release. Send "{" KeyToHoldDownPrev " up}" ; Release it. if KeyToHoldDown ; There is a key to press down. Send "{" KeyToHoldDown " down}" ; Press it down. } Joystick POV Hat The following example makes the joystick's POV hat behave like the arrow key cluster on a keyboard; that is, the POV hat will send arrow keystrokes (left, right, up, and down): SetTimer WatchPOV, 5 WatchPOV() { static KeyToHoldDown := "" POV := GetKeyState("JoyPOV") ; Get position of the POV control. KeyToHoldDownPrev := KeyToHoldDown ; Prev now holds the key that was down before (if any). ; Some joysticks might have a smooth/continous POV rather than one in fixed increments. ; To support them all, use a range: if POV < 0 ; No angle to report KeyToHoldDown := "" else if POV > 31500 ; 315 to 360 degrees: Forward KeyToHoldDown := "Up" else if POV >= 0 and POV <= 4500 ; 0 to 45 degrees: Forward KeyToHoldDown := "Up" else if POV >= 4501 and POV <= 13500 ; 45 to 135 degrees: Right KeyToHoldDown := "Right" else if POV >= 13501 and POV <= 22500 ; 135 to 225 degrees: Down KeyToHoldDown := "Down" else ; 225 to 315 degrees: Left KeyToHoldDown := "Left" if KeyToHoldDown = KeyToHoldDownPrev ; The correct key is already down (or no key is needed). return ; Do nothing. ; Otherwise, release the previous key and press down the new key: SetKeyDelay -1 ; Avoid delays between keystrokes. if KeyToHoldDownPrev ; There is a previous key to release. Send "{" KeyToHoldDownPrev " up}" ; Release it. if KeyToHoldDown ; There is a key to press down. Send "{" KeyToHoldDown " down}" ; Press it down. } Auto-repeating a Keystroke Both examples above can be modified to send the key repeatedly rather than merely holding it down (that is, they can mimic physically holding down a key on the keyboard). To do this, replace the following line: return ; Do nothing. With the following: { if KeyToHoldDown Send "{" KeyToHoldDown " down}" ; Auto-repeat the keystroke. return } Remarks A joystick other than first may be used by preceding the button or axis name with the number of the joystick. For example, 2Joy1 would be the second joystick's first button. To find other useful joystick scripts, visit the AutoHotkey forum. A keyword search such as Joystick and GetKeyState and Send is likely to produce topics of interest. Related Topics Joystick-To-Mouse script (using a joystick as a mouse) List of joystick buttons, axes, and controls GetKeyState Remapping the keyboard and mouse Windows Registry Editor Version 5.00 [-HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.ahk\UserChoice] PostMessage / SendMessage Tutorial | AutoHotkey v2 PostMessage / SendMessage Tutorial by Rajat This page explains how to send messages to a window or its controls via PostMessage or SendMessage and will answer some questions like: "How do I press a button on a minimized window?" "How do I select a menu item when MenuSelect doesn't work with it?!" "This is a skinnable*

window.... how do I send a command that works every time?" "and what about hidden windows?!" Requirements: Winspector Spy (can be found here) As a first example, note that MenuSelect fails to work with the menu bar on Outlook Express's "New Message" window. In other words, this code will not work: MenuSelect "New Message",, "&Insert", "&Picture..." But PostMessage can get the job done: PostMessage 0x0111, 40239, 0, , "New Message" Works like a charm! But what the heck is that? 0x0111 is the hex code of wm_command message and 40239 is the code that this particular window understands as menu-item 'Insert Picture' selection. Now let me tell you how to find a value such as 40239: Open Winspector Spy and a "New Message" window. Drag the crosshair from Winspector Spy's window to "New Message" window's titlebar (the portion not covered by Winspector Spy's overlay). Right click the selected window in the list on left and select 'Messages'. Right click the blank window and select 'Edit message filter'. Press the 'filter all' button and then dbl click 'wm_command' on the list on left. This way you will only monitor this message. Now go to the "New Message" window and select from its menu bar: Insert > Picture. Come back to Winspector Spy and press the traffic light button to pause monitoring. Expand the wm_command messages that've accumulated (forget others if any). What you want to look for (usually) is a code 0 message. sometimes there are wm_command messages saying 'win activated' or 'win destroyed' and other stuff.. not needed. You'll find that there's a message saying 'Control ID: 40239'...that's it! Now put that in the above function and you've got it! It's the wParam value. For the next example I'm taking Paint because possibly everyone will have that. Now let's say it's an app where you have to select a tool from a toolbar using AutoHotkey; say the dropper tool is to be selected. What will you do? Most probably a mouse click at the toolbar button, right? But toolbars can be moved and hidden! This one can be moved/hidden too. So if the target user has done any of this then your script will fail at that point. But the following function will still work: PostMessage 0x0111, 639,,, "Untitled - Paint" Another advantage to PostMessage is that the Window can be in the background; by contrast, sending mouse clicks would require it to be active. Here are some more examples. Note: I'm using WinXP Pro (SP1) ... if you use a different OS then your params may change (only applicable to apps like Wordpad and Notepad that come with windows; for others the params shouldn't vary): ;makes writing color teal in Wordpad PostMessage 0x0111, 32788, 0, , "Document - WordPad" ;opens about box in Notepad PostMessage 0x0111, 65, 0, , "Untitled - Notepad" ;toggles word-wrap in Notepad PostMessage 0x0111, 32, 0, , "Untitled - Notepad" ;play/pause in Windows Media Player PostMessage 0x0111, 32808, 0, , "Windows Media Player" ;suspend the hotkeys of a running AHK script DetectHiddenWindows True ; Use 65306 to Pause and 65303 to Reload instead of Suspend. (see FAQ) PostMessage 0x0111, 65305,,, "MyScript.ahk - AutoHotkey" ; Press CapsLock and Numpad2 to reload all AutoHotkey scripts CapsLock & Numpad2:: ReloadAllAhkScripts(ThisHotkey) { DetectHiddenWindows True for hwnd in WinGetList("ahk_class AutoHotkey") { if (hwnd = A_ScriptHwnd) ; ignore the current window for reloading continue PostMessage 0x0111, 65303,,, hwnd } Reload } This above was for PostMessage. SendMessage works the same way but additionally waits for a return value, which can be used for things such as getting the currently playing track in Winamp (see Automating Winamp for an example). Here are some more notes: The note above regarding OS being XP and msg values changing with different OSes is purely cautionary. if you've found a msg that works on your system (with a certain version of a software) then you can be sure it'll work on another system too with the same version of the software. In addition, most apps keep these msg values the same even on different versions of themselves (e.g. Windows Media Player and Winamp). If you've set the filter to show only wm_command in Winspector Spy and still you're getting a flood of messages then right click that message and select hide (msg name)... you don't want to have a look at a msg that appears without you interacting with the target software. The right pointing arrow in Winspector Spy shows the msg values and the blurred left pointing arrows show the returned value. A 0 (zero) value can by default safely be taken as 'no error' (use it with SendMessage). For posting to hidden windows add this to script: DetectHiddenWindows True Note: There are apps with which this technique doesn't work. I've had mixed luck with VB and Delphi apps. This technique is best used with C, C++ apps. With VB apps the 'LParam' of the same function keeps changing from one run to another. With Delphi apps... the GUI of some apps doesn't even use wm_command. It probably uses mouse pos & clicks. Go and explore.... and share your experiences in the AutoHotkey Forum. Feedback is welcome! This tutorial is not meant for total newbies (no offense meant) since these functions are considered advanced features. So if after reading the above you've not made heads or tails of it, please forget it. -Rajat List of Windows Messages | AutoHotkey v2 List of Windows Messages Below is a list of values for the Msg parameter of PostMessage and SendMessage. To discover more about how to use a particular message (e.g. WM_VSCROLL), look it up at Microsoft Docs or with a search engine of your choice. Also, check out the Message Tutorial. WM_NULL := 0x0000 WM_CREATE := 0x0001 WM_DESTROY := 0x0002 WM_MOVE := 0x0003 WM_SIZE := 0x0005 WM_ACTIVATE := 0x0006 WM_SETFOCUS := 0x0007 WM_KILLFOCUS := 0x0008 WM_ENABLE := 0x000A WM_SETREDRAW := 0x000B WM_SETTEXT := 0x000C WM_GETTEXT := 0x000D WM_GETTEXTLENGTH := 0x000E WM_PAINT := 0x000F WM_CLOSE := 0x0010 WM_QUERYENDSESSION := 0x0011 WM_QUIT := 0x0012 WM_QUERYOPEN := 0x0013 WM_ERASEBKGND := 0x0014 WM_SYSCOLORCHANGE := 0x0015 WM_ENDSESSION := 0x0016 WM_SYSTEMERROR := 0x0017 WM_SHOWWINDOW := 0x0018 WM_CTLCOLOR := 0x0019 WM_WININICHANGE := 0x001A WM_SETTINGCHANGE := 0x001A WM_DEVMODECHANGE := 0x001B WM_ACTIVATEAPP := 0x001C WM_FONTCHANGE := 0x001D WM_TIMECHANGE := 0x001E WM_CANCELMODE := 0x001F WM_SETCURSOR := 0x0020 WM_MOUSEACTIVATE := 0x0021 WM_CHILDACTIVATE := 0x0022 WM_QUEUESYNC := 0x0023 WM_GETMINMAXINFO := 0x0024 WM_PAINTICON := 0x0026 WM_ICONERASEBKGND := 0x0027 WM_NEXTDLGCTL := 0x0028 WM_SPOOLERSTATUS := 0x002A WM_DRAWITEM := 0x002B WM_MEASUREITEM := 0x002C WM_DELETEITEM := 0x002D WM_VKEYTOITEM := 0x002E WM_CHARTOITEM := 0x002F WM_SETFONT := 0x0030 WM_GETFONT := 0x0031 WM_SETHOTKEY := 0x0032 WM_GETHOTKEY := 0x0033 WM_QUERYDRAGICON := 0x0037 WM_COMPAREITEM := 0x0039 WM_COMPACTING := 0x0041 WM_WINDOWPOSCHANGING := 0x0046 WM_WINDOWPOSCHANGED := 0x0047 WM_POWER := 0x0048 WM_COPYDATA := 0x004A WM_CANCELJOURNAL := 0x004B WM_NOTIFY := 0x004E WM_INPUTLANGCHANGEREQUEST := 0x0050 WM_INPUTLANGCHANGE := 0x0051 WM_TCARD := 0x0052 WM_HELP := 0x0053 WM_USERCHANGED := 0x0054 WM_NOTIFYFORMAT := 0x0055 WM_CONTEXTMENU := 0x007B WM_STYLECHANGING := 0x007C WM_STYLECHANGED := 0x007D WM_DISPLAYCHANGE := 0x007E WM_GETICON := 0x007F WM_SETICON := 0x0080 WM_NCCREATE := 0x0081 WM_NCDESTROY := 0x0082 WM_NCCALCSIZE := 0x0083 WM_NCHITTEST := 0x0084 WM_NCPAINT := 0x0085 WM_NCACTIVATE := 0x0086 WM_GETDLGCODE := 0x0087 WM_NCMOUSEMOVE := 0x00A0 WM_NCLBUTTONDOWN := 0x00A1 WM_NCLBUTTONUP := 0x00A2 WM_NCLBUTTONDBLCLK := 0x00A3 WM_NCRBUTTONDOWN := 0x00A4 WM_NCRBUTTONUP := 0x00A5 WM_NCRBUTTONDBLCLK := 0x00A6 WM_NCMBUTTONDOWN := 0x00A7 WM_NCMBUTTONUP := 0x00A8 WM_NCMBUTTONDBLCLK := 0x00A9 WM_KEYFIRST := 0x0100 WM_KEYDOWN := 0x0100 WM_KEYUP := 0x0101 WM_CHAR := 0x0102 WM_DEADCHAR := 0x0103 WM_SYSKEYDOWN := 0x0104 WM_SYSKEYUP := 0x0105 WM_SYSCHAR := 0x0106 WM_SYSDEADCHAR := 0x0107 WM_KEYLAST := 0x0108 WM_IME_STARTCOMPOSITION := 0x010D WM_IME_ENDCOMPOSITION := 0x010E WM_IME_COMPOSITION := 0x010F WM_IME_KEYLAST := 0x010F WM_INITDIALOG := 0x0110 WM_COMMAND := 0x0111 WM_SYSCOMMAND := 0x0112 WM_TIMER := 0x0113 WM_HSCROLL := 0x0114 WM_VSCROLL := 0x0115 WM_INITMENU := 0x0116 WM_INITMENUPOPUP := 0x0117 WM_MENUSELECT := 0x011F WM_MENUCHAR := 0x0120 WM_ENTERIDLE := 0x0121 WM_CTLCOLORMSGBOX := 0x0132 WM_CTLCOLOREDIT := 0x0133 WM_CTLCOLORLISTBOX := 0x0134 WM_CTLCOLORBTN := 0x0135 WM_CTLCOLORDLG := 0x0136 WM_CTLCOLORSCROLLBAR := 0x0137 WM_CTLCOLORSTATIC := 0x0138 WM_MOUSEFIRST := 0x0200 WM_MOUSEMOVE := 0x0200 WM_LBUTTONDOWN := 0x0201 WM_LBUTTONUP := 0x0202 WM_LBUTTONDBLCLK := 0x0203 WM_RBUTTONDOWN := 0x0204 WM_RBUTTONUP := 0x0205 WM_RBUTTONDBLCLK := 0x0206 WM_MBUTTONDOWN := 0x0207 WM_MBUTTONUP := 0x0208 WM_MBUTTONDBLCLK := 0x0209 WM_MOUSEWHEEL := 0x020A WM_MOUSEHWHEEL := 0x020E WM_PARENTNOTIFY := 0x0210 WM_ENTERMENULOOP := 0x0211 WM_EXITMENULOOP := 0x0212 WM_NEXTMENU := 0x0213 WM_SIZING := 0x0214 WM_CAPTURECHANGED := 0x0215 WM_MOVING := 0x0216 WM_POWERBROADCAST := 0x0218 WM_DEVICECHANGE := 0x0219 WM_MDICREATE := 0x0220 WM_MDIDESTROY := 0x0221 WM_MDIACTIVATE := 0x0222 WM_MDIRESTORE := 0x0223 WM_MDINEXT := 0x0224 WM_MDIMAXIMIZE := 0x0225 WM_MDITILE := 0x0226 WM_MDICASCADE := 0x0227 WM_MDIICONARRANGE := 0x0228 WM_MDIGETACTIVE := 0x0229 WM_MDISETMENU := 0x0230 WM_ENTERSIZEMOVE := 0x0231 WM_EXITSIZEMOVE := 0x0232 WM_DROPFILES := 0x0233 WM_MDIREFRESHMENU := 0x0234 WM_IME_SETCONTEXT := 0x0281 WM_IME_NOTIFY := 0x0282 WM_IME_CONTROL := 0x0283 WM_IME_COMPOSITIONFULL := 0x0284 WM_IME_SELECT := 0x0285 WM_IME_CHAR := 0x0286 WM_IME_KEYDOWN := 0x0290 WM_IME_KEYUP := 0x0291 WM_MOUSEHOVER := 0x02A1 WM_NCMOUSELEAVE := 0x02A2 WM_MOUSELEAVE := 0x02A3 WM_CUT := 0x0300 WM_COPY := 0x0301 WM_PASTE := 0x0302 WM_CLEAR := 0x0303 WM_UNDO := 0x0304 WM_RENDERFORMAT := 0x0305 WM_RENDERALLFORMATS := 0x0306 WM_DESTROYCLIPBOARD := 0x0307 WM_DRAWCLIPBOARD := 0x0308 WM_PAINTCLIPBOARD := 0x0309 WM_VSCROLLCLIPBOARD := 0x030A WM_SIZECLIPBOARD := 0x030B WM_ASKCBFORMATNAME := 0x030C WM_CHANGECBCHAIN := 0x030D WM_HSCROLLCLIPBOARD := 0x030E WM_QUERYNEWPALETTE := 0x030F WM_PALETTEISCHANGING := 0x0310 WM_PALETTECHANGED := 0x0311 WM_HOTKEY := 0x0312 WM_PRINT := 0x0317 WM_PRINTCLIENT := 0x0318 WM_HANDHELDFIRST := 0x0358 WM_HANDHELDLAST := 0x035F WM_PENWINFIRST := 0x0380 WM_PENWINLAST := 0x038F WM_COALESCE_FIRST := 0x0390 WM_COALESCE_LAST := 0x039F WM_DDE_FIRST := 0x03E0 WM_DDE_INITIATE := 0x03E0 WM_DDE_TERMINATE := 0x03E1 WM_DDE_ADVISE := 0x03E2 WM_DDE_UNADVISE := 0x03E3 WM_DDE_ACK := 0x03E4 WM_DDE_DATA := 0x03E5 WM_DDE_REQUEST := 0x03E6 WM_DDE_POKE := 0x03E7 WM_DDE_EXECUTE := 0x03E8 WM_DDE_LAST := 0x03E8 WM_USER := 0x0400 WM_APP := 0x8000 Window and Control Styles | AutoHotkey v2 Window and Control Styles This page lists some styles and extended styles which can be set or retrieved with the methods Gui.Opt and GuiControl.Opt, and with the built-in functions WinSetStyle, WinSetExStyle, WinGetStyle, WinGetExStyle, ControlSetStyle, ControlSetExStyle, ControlGetStyle and ControlGetExStyle. Table of Contents Styles Common to Gui/Parent Windows and Most Control Types Text | Edit | UpDown | Picture Button | Checkbox | Radio | GroupBox DropDownList | ComboBox ListBox | ListView | TreeView DateTime | MonthCal Slider | Progress | Tab | StatusBar Common Styles By default, a GUI window uses WS_POPUP, WS_CAPTION, WS_SYSMENU, and WS_MINIMIZEBOX. For a GUI window, WS_CLIPSIBLINGS is always enabled and cannot be disabled. Style Hex Description WS_BORDER 0x800000 +/-Border. Creates a window that has a thin-line border. WS_POPUP 0x80000000 Creates a pop-up window. This style cannot be used with the

WS_CHILD style. WS_CAPTION 0xC00000 +/-Caption. Creates a window that has a title bar. This style is a numerical combination of WS_BORDER and WS_DLGFRAME. WS_CLIPSIBLINGS 0x4000000 Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window. WS_DISABLED 0x8000000 +/-Disabled. Creates a window that is initially disabled. WS_DLGFRAME 0x400000 Creates a window that has a border of a style typically used with dialog boxes. WS_GROUP 0x20000 +/-Group. Indicates that this control is the first one in a group of controls. This style is automatically applied to manage the "only one at a time" behavior of radio buttons. In the rare case where two groups of radio buttons are added consecutively (with no other control types in between them), this style may be applied manually to the first control of the second radio group, which splits it off from the first. WS_HSCROLL 0x100000 Creates a window that has a horizontal scroll bar. WS_MAXIMIZE 0x1000000 Creates a window that is initially maximized. WS_MAXIMIZEBOX 0x10000 +/-MaximizeBox. Creates a window that has a maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified. WS_MINIMIZE 0x20000000 Creates a window that is initially minimized. WS_MINIMIZEBOX 0x20000 +/-MinimizeBox. Creates a window that has a minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified. WS_OVERLAPPED 0x0 Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style. WS_OVERLAPPEDWINDOW 0xCF0000 Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style. WS_POPUPWINDOW 0x80880000 Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible. WS_SIZEBOX 0x40000 +/-Resize. Creates a window that has a sizing border. Same as the WS_THICKFRAME style. WS_SYSMENU 0x80000 +/-SysMenu. Creates a window that has a window menu on its title bar. The WS_CAPTION style must also be specified. WS_TABSTOP 0x10000 +/-Tabstop. Specifies a control that can receive the keyboard focus when the user presses Tab. Pressing Tab changes the keyboard focus to the next control with the WS_TABSTOP style. WS_THICKFRAME 0x40000 Creates a window that has a sizing border. Same as the WS_SIZEBOX style. WS_VSCROLL 0x200000 Creates a window that has a vertical scroll bar. WS_VISIBLE 0x10000000 Creates a window that is initially visible. WS_CHILD 0x40000000 Creates a child window. A window with this style cannot have a menu bar. This style cannot be used with the WS_POPUP style. Text Control Styles These styles affect the Text control. It has neither default styles nor forced styles. Style Hex Description SS_BLACKFRAME 0x7 Specifies a box with a frame drawn in the same color as the window frames. This color is black in the default color scheme. SS_BLACKRECT 0x4 Specifies a rectangle filled with the current window frame color. This color is black in the default color scheme. SS_CENTER 0x1 +/-Center. Specifies a simple rectangle and centers the text in the rectangle. The control automatically wraps words that extend past the end of a line to the beginning of the next centered line. SS_ETCHEDFRAME 0x12 Draws the frame of the static control using the EDGE_ETCHED edge style. SS_ETCHEDHORZ 0x10 Draws the top and bottom edges of the static control using the EDGE_ETCHED edge style. SS_ETCHEDVERT 0x11 Draws the left and right edges of the static control using the EDGE_ETCHED edge style. SS_GRAYFRAME 0x8 Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default color scheme. SS_GRAYRECT 0x5 Specifies a rectangle filled with the current screen background color. This color is gray in the default color scheme. SS_LEFT 0x0 +/-Left. This is the default. It specifies a simple rectangle and left-aligns the text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next left-aligned line. Words that are longer than the width of the control are truncated. SS_LEFTNOWORDWRAP 0xC +/-Wrap. Specifies a rectangle and left-aligns the text in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped. SS_NOPREFIX 0x80 Prevents interpretation of any ampersand (&) characters in the control's text as accelerator prefix characters. This can be useful when file names or other strings that might contain an ampersand (&) must be displayed within a text control. SS_NOTIFY 0x100 Sends the parent window the STN_CLICKED notification when the user clicks the control. SS_RIGHT 0x2 +/-Right. Specifies a rectangle and right-aligns the specified text in the rectangle. SS_SUNKEN 0x1000 Draws a half-sunken border around a static control. SS_WHITEFRAME 0x9 Specifies a box with a frame drawn with the same color as the window background. This color is white in the default color scheme. SS_WHITERECT 0x6 Specifies a rectangle filled with the current window background color. This color is white in the default color scheme. Edit Control Styles These styles affect the Edit control. By default, it uses WS_TABSTOP and WS_EX_CLIENTEDGE (extended style E0x200). It has no forced styles. If an Edit control is auto-detected as multi-line due to its starting contents containing multiple lines, its height being taller than 1 row, or its row-count having been explicitly specified as greater than 1, the following styles will be applied by default: WS_VSCROLL, ES_WANTRETURN, and ES_AUTOVSCROLL. If an Edit control is auto-detected as a single line, it defaults to having ES_AUTOHSCROLL. Style Hex Description ES_AUTOHSCROLL 0x80 +/-Wrap for multi-line edits, and +/-Limit for single-line edits. Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses Enter, the control scrolls all text back to the zero position. ES_AUTOVSCROLL 0x40 Scrolls text up one page when the user presses Enter on the last line. ES_CENTER 0x1 +/-Center. Centers text in a multiline edit control. ES_LOWERCASE 0x10 +/-Lowercase. Converts all characters to lowercase as they are typed into the edit control. ES_NOHIDESEL 0x100 Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus. ES_NUMBER 0x2000 +/-Number. Prevents the user from typing anything other than digits in the control. ES_OEMCONVERT 0x400 This style is most useful for edit controls that contain file names. ES_MULTILINE 0x4 +/-Multi. Designates a multiline edit control. The default is a single-line edit control. ES_PASSWORD 0x20 +/-Password. Displays a masking character in place of each character that is typed into the edit control, which conceals the text. ES_READONLY 0x800 +/-ReadOnly. Prevents the user from typing or editing text in the edit control. ES_RIGHT 0x2 +/-Right. Right-aligns text in a multiline edit control. ES_UPPERCASE 0x8 +/-Uppercase. Converts all characters to uppercase as they are typed into the edit control. ES_WANTRETURN 0x1000 +/-WantReturn. Specifies that a carriage return be inserted when the user presses Enter while typing text into a multiline edit control in a dialog box. If you do not specify this style, pressing Enter has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control. UpDown Control Styles These styles affect the UpDown control. By default, it uses UDS_ARROWKEYS, UDS_ALIGNRIGHT, UDS_SETBUDDYINT, and UDS_AUTOBUDDY. It has no forced styles. Style Hex Description UDS_WRAP 0x1 Named option "Wrap". Causes the control to wrap around to the other end of its range when the user attempts to go beyond the minimum or maximum. Without Wrap, the control stops when the minimum or maximum is reached. UDS_SETBUDDYINT 0x2 Causes the UpDown control to set the text of the buddy control (using the WM_SETTEXT message) when the position changes. However, if the buddy is a ListBox, the ListBox's current selection is changed instead. UDS_ALIGNRIGHT 0x4 Named option "Right" (default). Positions UpDown on the right side of its buddy control. UDS_ALIGNLEFT 0x8 Named option "Left". Positions UpDown on the left side of its buddy control. UDS_AUTOBUDDY 0x10 Automatically selects the previous control in the z-order as the UpDown control's buddy control. UDS_ARROWKEYS 0x20 Allows the user to press â†‘ or â†“ on the keyboard to increase or decrease the UpDown control's position. UDS_HORZ 0x40 Named option "Horz". Causes the control's arrows to point left and right instead of up and down. UDS_NOTHOUSANDS 0x80 Does not insert a thousands separator between every three decimal digits in the buddy control. UDS_HOTTRACK 0x100 Causes the control to exhibit "hot tracking" behavior. That is, it highlights the control's buttons as the mouse passes over them. This flag may be ignored if the desktop theme overrides it. Picture Control Styles These styles affect the Picture control. It has no default styles. The style SS_ICON (for icons and cursors) or SS_BITMAP (for other image types) is always enabled and cannot be disabled. Style Hex Description SS_REALSIZECONTROL 0x40 Adjusts the bitmap to fit the size of the control. SS_CENTERIMAGE 0x200 Centers the bitmap in the control. If the bitmap is too large, it will be clipped. For text controls, if the control contains a single line of text, the text is centered vertically within the available height of the control. Button, CheckBox, Radio, and GroupBox Control Styles These styles affect the Button, CheckBox, Radio, or GroupBox controls. By default, each of these controls except GroupBox uses the styles BS_MULTILINE (unless it has no explicitly set width or height, nor any CR/LF characters in their text) and WS_TABSTOP (however, Radio controls other than the first of each radio group lack WS_TABSTOP). In addition, Radio controls have BS_NOTIFY so that double clicks can be detected. The following styles are always enabled and cannot be disabled: Button: BS_PUSHBUTTON or BS_DEFPUSHBUTTON CheckBox: BS_AUTOCHECKBOX or BS_AUTO3STATE Radio: BS_AUTORADIOBUTTON GroupBox: BS_GROUPBOX Style Hex Description BS_AUTO3STATE 0x6 Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, indeterminate, and cleared. BS_AUTOCHECKBOX 0x3 Creates a button that is the same as a check box, except that the check state automatically toggles between checked and cleared each time the user selects the check box. BS_AUTORADIOBUTTON 0x9 Creates a button that is the same as a radio button, except that when the user selects it, the system automatically sets the button's check state to checked and automatically sets the check state for all other buttons in the same group to cleared. BS_LEFT 0x100 +/-Left. Left-aligns the text. BS_PUSHBUTTON 0x0 Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button. BS_PUSHLIKE 0x1000 Makes a checkbox or radio button look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked. BS_RIGHT 0x200 +/-Right. Right-aligns the text. BS_RIGHTBUTTON 0x20 +Right (i.e. +Right includes both BS_RIGHT and BS_RIGHTBUTTON, but -Right removes only BS_RIGHT, not BS_RIGHTBUTTON). Positions a checkbox square or radio button circle on the right side of the control's available width instead of the left. BS_BOTTOM 0x800 Places the text at the bottom of the control's available height. BS_CENTER 0x300 +/-Center. Centers the text horizontally within the control's available width. BS_DEFPUSHBUTTON 0x1 +/-Default. Creates a push button with a heavy black border. If the button is in a dialog box, the user can select the button by pressing Enter, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely option. BS_MULTILINE 0x2000 +/-Wrap. Wraps the text to multiple lines if the text is too long to fit on a single line in the control's available width. This also allows linefeed (`n) to start new lines of text. BS_NOTIFY 0x4000 Enables a button to send BN_KILLFOCUS and BN_SETFOCUS notification codes to its parent window. Note that buttons send the BN_CLICKED notification code regardless of whether it has this style. To get BN_DBLCLK notification codes, the button must have the BS_RADIOBUTTON or BS_OWNERDRAW style. BS_TOP 0x400 Places text at the top of the control's available height. BS_VCENTER 0xC00 Vertically centers

text in the control's available height. BS_FLAT 0x8000 Specifies that the button is two-dimensional; it does not use the default shading to create a 3-D effect. BS_GROUPBOX 0x7 Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper left corner. DropDownList and ComboBox Control Styles These styles affect the DropDownList and ComboBox controls. By default, each of these controls uses WS_TABSTOP. In addition, a DropDownList control uses WS_VSCROLL, and a ComboBox control uses WS_VSCROLL and CBS_AUTOHSCROLL. The following styles are always enabled and cannot be disabled: DropDownList: CBS_DROPDOWNLIST ComboBox: Either CBS_DROPDOWN or CBS_SIMPLE Style Hex Description CBS_AUTOHSCROLL 0x40 +/-Limit. Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is enabled. CBS_DISABLENOSCROLL 0x800 Shows a disabled vertical scroll bar in the drop-down list when it does not contain enough items to scroll. Without this style, the scroll bar is hidden when the drop-down list does not contain enough items. CBS_DROPDOWN 0x2 Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the edit control. CBS_DROPDOWNLIST 0x3 Similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item that displays the current selection in the list box. CBS_LOWERCASE 0x4000 +/-Lowercase. Converts to lowercase any uppercase characters that are typed into the edit control of a combo box. CBS_NOINTEGRALHEIGHT 0x400 Specifies that the combo box will be exactly the size specified by the application when it created the combo box. Usually, Windows CE sizes a combo box so that it does not display partial items. CBS_OEMCONVERT 0x80 Converts text typed in the combo box edit control from the Windows CE character set to the OEM character set and then back to the Windows CE set. This style is most useful for combo boxes that contain file names. It applies only to combo boxes created with the CBS_DROPDOWN style. CBS_SIMPLE 0x1 +/-Simple (ComboBox only). Displays the drop-down list at all times. The current selection in the list is displayed in the edit control. CBS_SORT 0x100 +/-Sort. Sorts the items in the drop-list alphabetically. CBS_UPPERCASE 0x2000 +/-Uppercase. Converts to uppercase any lowercase characters that are typed into the edit control of a ComboBox. ListBox Control Styles These styles affect the ListBox control. By default, it uses WS_TABSTOP, LBS_USETABSTOPS, WS_VSCROLL, and WS_EX_CLIENTEDGE (extended style E0x200). The style LBS_NOTIFY (supports detection of double-clicks) is always enabled and cannot be disabled. Style Hex Description LBS_DISABLENOSCROLL 0x1000 Shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If you do not specify this style, the scroll bar is hidden when the list box does not contain enough items. LBS_NOINTEGRALHEIGHT 0x100 Specifies that the list box will be exactly the size specified by the application when it created the list box. LBS_EXTENDEDSEL 0x800 +/-Multi. Allows multiple selections via control-click and shift-click. LBS_MULTIPLESEL 0x8 A simplified version of multi-select in which control-click and shift-click are not necessary because normal left clicks serve to extend the selection or de-select a selected item. LBS_NOSEL 0x4000 +/-ReadOnly. Specifies that the user can view list box strings but cannot select them. LBS_NOTIFY 0x1 Causes the list box to send a notification code to the parent window whenever the user clicks a list box item (LBN_SELCHANGE), double-clicks an item (LBN_DBLCLK), or cancels the selection (LBN_SELCANCEL). LBS_SORT 0x2 +/-Sort. Sorts the items in the list box alphabetically. LBS_USETABSTOPS 0x80 Enables a ListBox to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units apart. A dialog box unit is equal to one-fourth of the current dialog box base-width unit. ListView Control Styles These styles affect the ListView control. By default, it uses WS_TABSTOP, LVS_REPORT, LVS_SHOWSELALWAYS, LVS_EX_FULLROWSELECT, LVS_EX_HEADERDRAGDROP, and WS_EX_CLIENTEDGE (extended style E0x200). It has no forced styles. Style Hex Description LVS_ALIGNLEFT 0x800 Items are left-aligned in icon and small icon view. LVS_ALIGNTOP 0x0 Items are aligned with the top of the list-view control in icon and small icon view. This is the default. LVS_AUTOARRANGE 0x100 Icons are automatically kept arranged in icon and small icon view. LVS_EDITLABELS 0x200 +/-ReadOnly. Specifying -ReadOnly (or +0x200) allows the user to edit the first field of each row in place. LVS_ICON 0x0 +Icon. Specifies large-icon view. LVS_LIST 0x3 +List. Specifies list view. LVS_NOCOLUMNHEADER 0x4000 +/-Hdr. Avoids displaying column headers in report view. LVS_NOLABELWRAP 0x80 Item text is displayed on a single line in icon view. By default, item text may wrap in icon view. LVS_NOSCROLL 0x2000 Scrolling is disabled. All items must be within the client area. This style is not compatible with the LVS_LIST or LVS_REPORT styles. LVS_NOSORTHEADER 0x8000 +/-NoSortHdr. Column headers do not work like buttons. This style can be used if clicking a column header in report view does not carry out an action, such as sorting. LVS_OWNERDATA 0x1000 This style specifies a virtual list-view control (not directly supported by AutoHotkey). LVS_OWNERDRAWFIXED 0x400 The owner window can paint items in report view in response to WM_DRAWITEM messages (not directly supported by AutoHotkey). LVS_REPORT 0x1 +Report. Specifies report view. LVS_SHAREIMAGELISTS 0x40 The image list will not be deleted when the control is destroyed. This style enables the use of the same image lists with multiple list-view controls. LVS_SHOWSELALWAYS 0x8 The selection, if any, is always shown, even if the control does not have keyboard focus. LVS_SINGLESEL 0x4 +/-Multi. Only one item at a time can be selected. By default, multiple items can be selected. LVS_SMALLICON 0x2 +IconSmall. Specifies small-icon view. LVS_SORTASCENDING 0x10 +/-Sort. Rows are sorted in ascending order based on the contents of the first field. LVS_SORTDESCENDING 0x20 +/-SortDesc. Same as above but in descending order. Extended ListView styles require the LV prefix when used with the Gui methods/properties. Some extended styles introduced in Windows XP or later versions are not listed here. For a full list, see Microsoft Docs: Extended List-View Styles. Extended Style Hex Description LVS_EX_BORDERSELECT LV0x8000 When an item is selected, the border color of the item changes rather than the item being highlighted (might be non-functional in recent operating systems). LVS_EX_CHECKBOXES LV0x4 +/-Checked. Displays a checkbox with each item. When set to this style, the control creates and sets a state image list with two images using DrawFrameControl. State image 1 is the unchecked box, and state image 2 is the checked box. Setting the state image to zero removes the check box altogether. Checkboxes are visible and functional with all list-view modes except the tile view mode. Clicking a checkbox in tile view mode only selects the item; the state does not change. LVS_EX_DOUBLEBUFFER LV0x10000 Paints via double-buffering, which reduces flicker. This extended style also enables alpha-blended marquee selection on systems where it is supported. LVS_EX_FLATSB LV0x100 Enables flat scroll bars in the list view. LVS_EX_FULLROWSELECT LV0x20 When a row is selected, all its fields are highlighted. This style is available only in conjunction with the LVS_REPORT style. LVS_EX_GRIDLINES LV0x1 +/-Grid. Displays gridlines around rows and columns. This style is available only in conjunction with the LVS_REPORT style. LVS_EX_HEADERDRAGDROP LV0x10 Enables drag-and-drop reordering of columns in a list-view control. This style is only available to list-view controls that use the LVS_REPORT style. LVS_EX_INFOTIP LV0x400 When a list-view control uses this style, the LVN_GETINFOTIP notification message is sent to the parent window before displaying an item's ToolTip. LVS_EX_LABELTIP LV0x4000 If a partially hidden label in any list-view mode lacks ToolTip text, the list-view control will unfold the label. If this style is not set, the list-view control will unfold partly hidden labels only for the large icon mode. Note: On some versions of Windows, this style might not work properly if the GUI window is set to be always-on-top. LVS_EX_MULTIWORKAREAS LV0x2000 If the list-view control has the LVS_AUTOARRANGE style, the control will not autoarrange its icons until one or more work areas are defined (see LVM_SETWORKAREAS). To be effective, this style must be set before any work areas are defined and any items have been added to the control. LVS_EX_ONECLICKACTIVATE LV0x40 The list-view control sends an LVN_ITEMACTIVATE notification message to the parent window when the user clicks an item. This style also enables hot tracking in the list-view control. Hot tracking means that when the cursor moves over an item, it is highlighted but not selected. LVS_EX_REGIONAL LV0x200 Sets the list-view window region to include only the item icons and text using SetWindowRgn. Any area that is not part of an item is excluded from the window region. This style is only available to list-view controls that use the LVS_ICON style. LVS_EX_SIMPLESELECT LV0x100000 In icon view, moves the state image of the item to the top right of the large icon rendering. In views other than icon view there is no change. When the user changes the state by using the space bar, all selected items cycle over, not the item with the focus. LVS_EX_SUBITEMIMAGES LV0x2 Allows images to be displayed for fields beyond the first. This style is available only in conjunction with the LVS_REPORT style. LVS_EX_TRACKSELECT LV0x8 Enables hot-track selection in a list-view control. Hot track selection means that an item is automatically selected when the cursor remains over the item for a certain period of time. The delay can be changed from the default system setting with a LVM_SETHOVERTIME message. This style applies to all styles of list-view control. You can check whether hot-track selection is enabled by calling SystemParametersInfo. LVS_EX_TWOCLICKACTIVATE LV0x80 The list-view control sends an LVN_ITEMACTIVATE notification message to the parent window when the user double-clicks an item. This style also enables hot tracking in the list-view control. Hot tracking means that when the cursor moves over an item, it is highlighted but not selected. LVS_EX_UNDERLINECOLD LV0x1000 Causes those non-hot items that may be activated to be displayed with underlined text. This style requires that LVS_EX_TWOCLICKACTIVATE be set also. LVS_EX_UNDERLINEHOT LV0x800 Causes those hot items that may be activated to be displayed with underlined text. This style requires that LVS_EX_ONECLICKACTIVATE or LVS_EX_TWOCLICKACTIVATE also be set. TreeView Control Styles These styles affect the TreeView control. By default, it uses WS_TABSTOP, TVS_SHOWSELALWAYS, TVS_HASLINES, TVS_LINESATROOT, TVS_HASBUTTONS, and WS_EX_CLIENTEDGE (extended style E0x200). It has no forced styles. Style Hex Description TVS_CHECKBOXES 0x100 +/-Checked. Displays a checkbox next to each item. TVS_DISABLEDRAGDROP 0x10 Prevents the tree-view control from sending TVN_BEGINDRAG notification messages. TVS_EDITLABELS 0x8 +/-ReadOnly. Allows the user to edit the names of tree-view items. TVS_FULLROWSELECT 0x1000 Enables full-row selection in the tree view. The entire row of the selected item is highlighted, and clicking anywhere on an item's row causes it to be selected. This style cannot be used in conjunction with the TVS_HASLINES style. TVS_HASBUTTONS 0x1 +/-Buttons. Displays plus (+) and minus (-) buttons next to parent items. The user clicks the buttons to expand or collapse a parent item's list of child items. To include buttons with items at the root of the tree view, TVS_LINESATROOT must also be specified. TVS_HASLINES 0x2 +/-Lines. Uses lines to show the hierarchy of items. TVS_INFOTIP 0x800 Obtains ToolTip information by sending the TVN_GETINFOTIP notification. TVS_LINESATROOT 0x4 +/-Lines. Uses lines to link items at the root of the tree-view control. This value is ignored if TVS_HASLINES is not also specified. TVS_NOHSCROLL 0x8000 +/-HScroll. Disables horizontal scrolling in the control. The control will not display any horizontal scroll bars. TVS_NONEVENHEIGHT 0x4000 Sets the height of the items to an odd height with the TVM_SETITEMHEIGHT message. By default, the height of items must be an even value. TVS_NOSCROLL 0x2000 Disables both horizontal and vertical scrolling in the control. The control will not display any scroll bars. TVS_NOTOOLTIPS 0x80 Disables tooltips. TVS_RTLREADING 0x40 Causes text to be displayed from right-to-left (RTL). Usually, windows display text left-to-right (LTR). TVS_SHOWSELALWAYS 0x20 Causes a selected item to remain selected when the tree-view control loses focus. TVS_SINGLEEXPAND 0x400 Causes the item being selected to expand and the

item being unselected to collapse upon selection in the tree-view. If the user holds down Ctrl while selecting an item, the item being unselected will not be collapsed. TVS_TRACKSELECT 0x200 Enables hot tracking of the mouse in a tree-view control. DateTime Control Styles These styles affect the DateTime control. By default, it uses DTS_SHORTDATECENTURYFORMAT and WS_TABSTOP. It has no forced styles. Style Hex Description DTS_UPDOWN 0x1 Provides an up-down control to the right of the control to modify date-time values, which replaces the of the drop-down month calendar that would otherwise be available. DTS_SHOWNONE 0x2 Displays a checkbox inside the control that users can uncheck to make the control have no date/time selected. Whenever the control has no date/time, Gui.Submit and GuiCtrl.Value will retrieve a blank value (empty string). DTS_SHORTDATEFORMAT 0x0 Displays the date in short format. In some locales, it looks like 6/1/05 or 6/1/2005. On older operating systems, a two-digit year might be displayed. This is why DTS_SHORTDATECENTURYFORMAT is the default and not DTS_SHORTDATEFORMAT. DTS_LONGDATEFORMAT 0x4 Format option "LongDate". Displays the date in long format. In some locales, it looks like Wednesday, June 01, 2005. DTS_SHORTDATECENTURYFORMAT 0xC Format option blank/omitted. Displays the date in short format with four-digit year. In some locales, it looks like 6/1/2005. If the system's version of Comctl32.dll is older than 5.8, this style is not supported and DTS_SHORTDATEFORMAT is automatically substituted. DTS_TIMEFORMAT 0x9 Format option "Time". Displays only the time, which in some locales looks like 5:31:42 PM. DTS_APPCANPARSE 0x10 Not yet supported. Allows the owner to parse user input and take necessary action. It enables users to edit within the client area of the control when they press F2. The control sends DTN_USERSTRING notification messages when users are finished. DTS_RIGHTALIGN 0x20 +/-Right. The calendar will drop down on the right side of the control instead of the left. MonthCal Control Styles These styles affect the MonthCal control. By default, it uses WS_TABSTOP. It has no forced styles. Style Hex Description MCS_DAYSTATE 0x1 Makes the control send MCN_GETDAYSTATE notifications to request information about which days should be displayed in bold. [Not yet supported] MCS_MULTISELECT 0x2 Named option "Multi". Allows the user to select a range of dates rather than being limited to a single date. By default, the maximum range is 366 days, which can be changed by sending the MCM_SETMAXSELCOUNT message to the control. For example: SendMessage 0x1004, 7, 0, "SysMonthCal321", MyGui ; 7 days. 0x1004 is MCM_SETMAXSELCOUNT. MCS_WEEKNUMBERS 0x4 Displays week numbers (1-52) to the left of each row of days. Week 1 is defined as the first week that contains at least four days. MCS_NOTODAYCIRCLE 0x8 Prevents the circling of today's date within the control. MCS_NOTODAY 0x10 Prevents the display of today's date at the bottom of the control. Slider Control Styles These styles affect the Slider control. By default, it uses WS_TABSTOP. It has no forced styles. Style Hex Description TBS_VERT 0x2 +/-Vertical. The control is oriented vertically. TBS_LEFT 0x4 +/-Left. The control displays tick marks at the top of the control (or to its left if TBS_VERT is present). Same as TBS_TOP. TBS_TOP 0x4 same as TBS_LEFT. TBS_BOTH 0x8 +/-Center. The control displays tick marks on both sides of the control. This will be both top and bottom when used with TBS_HORZ or both left and right if used with TBS_VERT. TBS_AUTOTICKS 0x1 The control has a tick mark for each increment in its range of values. Use +/-TickInterval to have more flexibility. TBS_ENABLESELRANGE 0x20 The control displays a selection range only. The tick marks at the starting and ending positions of a selection range are displayed as triangles (instead of vertical dashes), and the selection range is highlighted (highlighting might require that the theme be removed via GuiObj.Opt("-Theme")). To set the selection range, follow this example, which sets the starting position to 55 and the ending position to 66: SendMessage 0x040B, 1, 55, "msctls_trackbar321", WinTitle SendMessage 0x040C, 1, 66, "msctls_trackbar321", WinTitle TBS_FIXEDLENGTH 0x40 +/-Thick. Allows the thumb's size to be changed. TBS_NOTHUMB 0x80 The control does not display the moveable bar. TBS_NOTICKS 0x10 +/-NoTicks. The control does not display any tick marks. TBS_TOOLTIPS 0x100 +/-ToolTip. The control supports tooltips. When a control is created using this style, it automatically creates a default ToolTip control that displays the slider's current position. You can change where the tooltips are displayed by using the TBM_SETTIPSIDE message. TBS_REVERSED 0x200 Unfortunately, this style has no effect on the actual behavior of the control, so there is probably no point in using it (instead, use +Invert in the control's options to reverse it). Depending on OS version, this style might require Internet Explorer 5.0 or greater. TBS_DOWNISLEFT 0x400 Unfortunately, this style has no effect on the actual behavior of the control, so there is probably no point in using it. Depending on OS version, this style might require Internet Explorer 5.01 or greater. Progress Control Styles These styles affect the Progress control. By default, it uses PBS_SMOOTH. It has no forced styles. Style Hex Description PBS_SMOOTH 0x1 +/-Smooth. The progress bar displays progress status in a smooth scrolling bar instead of the default segmented bar. When this style is present, the control automatically reverts to the Classic Theme appearance. PBS_VERTICAL 0x4 +/-Vertical. The progress bar displays progress status vertically, from bottom to top. PBS_MARQUEE 0x8 The progress bar moves like a marquee; that is, each change to its position causes the bar to slide further along its available length until it wraps around to the other side. A bar with this style has no defined position. Each attempt to change its position will instead slide the bar by one increment. This style is typically used to indicate an ongoing operation whose completion time is unknown. Tab Control Styles These styles affect the Tab control. By default, it uses WS_TABSTOP and TCS_MULTILINE. The style WS_CLIPSIBLINGS is always enabled and cannot be disabled, while TCS_OWNERDRAWFIXED is forced on or off as required by the control's background color and/or text color. Style Hex Description TCS_SCROLLOPPOSITE 0x1 Unneeded tabs scroll to the opposite side of the control when a tab is selected. TCS_BOTTOM 0x2 +/-Bottom. Tabs appear at the bottom of the control instead of the top. TCS_RIGHT 0x2 Tabs appear vertically on the right side of controls that use the TCS_VERTICAL style. TCS_MULTISELECT 0x4 Multiple tabs can be selected by holding down Ctrl when clicking. This style must be used with the TCS_BUTTONS style. TCS_FLATBUTTONS 0x8 Selected tabs appear as being indented into the background while other tabs appear as being on the same plane as the background. This style only affects tab controls with the TCS_BUTTONS style. TCS_FORCEICONLEFT 0x10 Icons are aligned with the left edge of each fixed-width tab. This style can only be used with the TCS_FIXEDWIDTH style. TCS_FORCELABELLEFT 0x20 Labels are aligned with the left edge of each fixed-width tab; that is, the label is displayed immediately to the right of the icon instead of being centered. This style can only be used with the TCS_FIXEDWIDTH style, and it implies the TCS_FORCEICONLEFT style. TCS_HOTTRACK 0x40 Items under the pointer are automatically highlighted. TCS_VERTICAL 0x80 +/-Left or +/-Right. Tabs appear at the left side of the control, with tab text displayed vertically. This style is valid only when used with the TCS_MULTILINE style. To make tabs appear on the right side of the control, also use the TCS_RIGHT style. This style will not correctly display the tabs if a custom background color or text color is in effect. To workaround this, specify -Background and/or cDefault in the tab control's options. TCS_BUTTONS 0x100 +/-Buttons. Tabs appear as buttons, and no border is drawn around the display area. TCS_SINGLELINE 0x0 +/-Wrap. Only one row of tabs is displayed. The user can scroll to see more tabs, if necessary. This style is the default. TCS_MULTILINE 0x200 +/-Wrap. Multiple rows of tabs are displayed, if necessary, so all tabs are visible at once. TCS_RIGHTJUSTIFY 0x0 This is the default. The width of each tab is increased, if necessary, so that each row of tabs fills the entire width of the tab control. This window style is ignored unless the TCS_MULTILINE style is also specified. TCS_FIXEDWIDTH 0x400 All tabs are the same width. This style cannot be combined with the TCS_RIGHTJUSTIFY style. TCS_RAGGEDRIGHT 0x800 Rows of tabs will not be stretched to fill the entire width of the control. This style is the default. TCS_FOCUSONBUTTONDOWN 0x1000 The tab control receives the input focus when clicked. TCS_OWNERDRAWFIXED 0x2000 The parent window is responsible for drawing tabs. TCS_TOOLTIPS 0x4000 The tab control has a tooltip control associated with it. TCS_FOCUSNEVER 0x8000 The tab control does not receive the input focus when clicked. StatusBar Control Styles These styles affect the StatusBar control. By default, it uses SBARS_TOOLTIPS and SBARS_SIZEGRIP (the latter only if the window is resizable). It has no forced styles. Style Hex Description SBARS_TOOLTIPS 0x800 Displays a tooltip when the mouse hovers over a part of the status bar that: 1) has too much text to be fully displayed; or 2) has an icon but no text. The text of the tooltip can be set via: SendMessage 0x0411, 0, StrPtr("Text to display"), "msctls_statusbar321", MyGui ; 0x0411 is SB_SETTIPTEXTW. The bold 0 above is the zero-based part number. To use a part other than the first, specify 1 for second, 2 for the third, etc. NOTE: The tooltip might never appear on certain OS versions. SBARS_SIZEGRIP 0x100 Includes a sizing grip at the right end of the status bar. A sizing grip is similar to a sizing border; it is a rectangular area that the user can click and drag to resize the parent window. Threads - Behaviour & Priority | AutoHotkey v2 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. Automating Winamp | AutoHotkey v2 Automating Winamp This section demonstrates how to control Winamp via hotkey even when it is minimized or inactive. This information has been tested with Winamp 2.78c but should work for other major releases as well. Please post changes and improvements in the forum or contact the author. This example makes the Ctrl+Alt+P hotkey equivalent to pressing Winamp's pause/unpause button: ^!p:: { if not WinExist("ahk_class Winamp v1.x") return ; Otherwise, the above has set the "last found" window for use below. ControlSend "c" ; Pause/Unpause } Here are some of the keyboard shortcuts available in Winamp 2.x (may work in other versions too). The above example can be revised to use any of these keys: Key to send Effect c Pause/UnPause x Play/Restart/UnPause v Stop +v Stop with Fadeout ^v Stop after the current track b Next Track z Previous Track {left} Rewind 5 seconds {right} Fast-forward 5 seconds {up} Turn Volume Up {down} Turn Volume Down The following example asks Winamp which track number is currently active: TrackNumber := SendMessage(0x0400, 0, 120,, "ahk_class Winamp v1.x") if (TrackNumber != "") { TrackNumber += 1 ; Winamp's count starts at 0, so adjust by 1. MsgBox "Track #" TrackNumber " is active or playing." } WinTitle & Last Found Window | AutoHotkey v2 WinTitle Parameter & Last Found Window Various built-in functions have a WinTitle parameter, used to identify which window (or windows) to operate on. This parameter can be the title or partial title of the window, and/or any other criteria described on this page. Quick Reference: TitleMatching Behaviour IntegerHWND ObjectObject with HWND AThe Active Window ahk_classWindow Class ahk_idUnique ID/HWND ahk_pidProcess ID ahk_exeProcess Name/Path ahk_groupWindow Group Multiple Criteria (All empty)Last Found Window Matching Behaviour SetTitleMatchMode controls how a partial or complete title is compared against the title of each window. Depending on the setting, WinTitle can be an exact title, or it can contain a prefix, a substring which occurs anywhere in the title, or a RegEx pattern. This setting also controls whether the ahk_class and ahk_exe criteria are interpreted as RegEx patterns. Window titles are case sensitive, except when using the i) modifier in a RegEx pattern. Hidden windows are only detected if DetectHiddenWindows is turned on, except with WinShow, which always detects hidden windows. If multiple windows match WinTitle and any other criteria, the topmost matching window is used. If the active window matches the criteria, it usually takes precedence since it is usually above all other windows. However, if an always-on-top window also matches (and the active window is not always-on-top), it may be used instead. A (Active Window) Use the letter A for WinTitle and omit the other three window parameters (WinText, ExcludeTitle and ExcludeText), to operate on the active window. The following example retrieves and reports the unique ID (HWND) of the active window: MsgBox WinExist("A") The following example creates a hotkey which can be pressed to maximize the active window: #Up::WinMaximize "A" ; Win+Up ahk_ Criteria Specify one or more of the following ahk_ criteria (optionally in addition to a window's title). An ahk_ criterion always consists of an ahk_ keyword and its criterion value, both separated by zero or more spaces or tabs. For example, ahk_class Notepad represents a Notepad window. The ahk_ keyword and its criterion value do not need to be separated by spaces or tabs. This is primarily useful when specifying ahk_ criteria in combination with variables. For example, you could specify "ahk_pid" PID instead of "ahk_pid " PID. In all other cases, however, it is recommended to use at least one space or tab as a separation to improve readability. ahk_class (Window Class) Use ahk_class ClassName in WinTitle to identify a window by its window class. A window class is a set of attributes that the system uses as a template to create a window. In other words, the class name of the window identifies what type of window it is. A window class can be revealed via Window Spy or retrieved by WinGetClass. If the RegEx title matching mode is active, ClassName accepts a regular expression. The following example activates a console window (e.g. cmd.exe): WinActivate "ahk_class ConsoleWindowClass" The following example does the same as above, but uses a regular expression (note that the RegEx title matching mode must be turned on beforehand to make it work): WinActivate "ahk_class i)^ConsoleWindowClass$" ahk_id (Unique ID / HWND) Each window or control has a unique ID, also known as a HWND (short for handle to window). This ID can be used to identify the window or control even if its title changes. Use ahk_id HWND, HWND (that is, an Integer without the ahk_id keyword) or an Object with a HWND property in WinTitle to identify a window or control by its unique ID. DetectHiddenWindows affects whether the ahk_id criterion detects hidden top-level windows, but hidden controls are always detected. It also affects whether hidden windows identified by HWND (as an Object property or Integer) are detected, but only when used with WinWait or WinWaitClose. The ahk_id criterion can also be combined with other criteria that the given window must match. For example, WinExist("ahk_id " ahwnd " ahk_class " aclass) returns ahwnd if the window is "detected" (according to DetectHiddenWindows) and its window class matches the string contained by aclass. The ID of a window is typically retrieved via WinExist or WinGetID. The ID of a control is typically retrieved via ControlGetHwnd, MouseGetPos, or DllCall. Gui and GuiControl objects have a Hwnd property and therefore can be used directly in WinTitle. The following examples operate on a window by a unique ID stored in VarContainingID: ; Pass an Integer. WinActivate Integer(VarContainingID) WinShow A_ScriptHwnd WinWaitNotActive WinExist("A") ; Pass an Object with a HWND property. WinActivate {Hwnd: VarContainingID} WinWaitClose myGuiObject ; Use the ahk_id prefix. WinActivate "ahk_id " VarContainingID If the object has no HWND property or the property's value is not an integer, a PropertyError or TypeError is thrown. ahk_pid (Process ID) Use ahk_pid PID in WinTitle to identify a window belonging to a specific process. The process identifier (PID) is typically retrieved by WinGetPID, Run or Process functions. The ID of a window's process can be revealed via Window Spy. The following example activates a window by a process ID stored in VarContainingPID: WinActivate "ahk_pid " VarContainingPID ahk_exe (Process Name/Path) Use ahk_exe ProcessNameOrPath in WinTitle to identify a window belonging to any process with the given name or path. While the ahk_pid criterion is limited to one specific process, the ahk_exe criterion considers all processes with name or full path matching a given string. If the RegEx title matching mode is active, ProcessNameOrPath accepts a regular expression which must match the full path of the process. Otherwise, it accepts a case-insensitive name or full path; for example, ahk_exe notepad.exe covers ahk_exe C:\Windows\Notepad.exe, ahk_exe C:\Windows\System32\Notepad.exe and other variations. The name of a window's process can be revealed via Window Spy. The following example activates a Notepad window by its process name: WinActivate "ahk_exe notepad.exe" The following example does the same as above, but uses a regular expression (note that the RegEx title matching mode must be turned on beforehand to make it work): WinActivate "ahk_exe i)\\notepad\.exe$" ; Match the name part of the full path. ahk_group (Window Group) Use ahk_group GroupName in WinTitle to identify a window or windows matching the rules contained by a previously defined window group. WinMinimize, WinMaximize, WinRestore, WinHide, WinShow, WinClose, and WinKill will act on all the group's windows. By contrast, the other window functions such as WinActivate and WinExist will operate only on the topmost window of the group. The following example activates any window matching the criteria defined by a window group: ; Define the group: Windows Explorer windows GroupAdd "Explorer", "ahk_class ExploreWClass" ; Unused on Vista and later GroupAdd "Explorer", "ahk_class CabinetWClass" ; Activate any window matching the above criteria WinActivate "ahk_group Explorer" Multiple Criteria By contrast with the ahk_group criterion (which broadens the search), the search may be narrowed by specifying more than one criterion within the WinTitle parameter. In the following example, the script waits for a window whose title contains My File.txt and whose class is Notepad: WinWait "My File.txt ahk_class Notepad" WinActivate ; Activate the window it found. When using this method, the text of the title (if any is desired) should be listed first, followed by one or more additional criteria. Criteria beyond the first should be separated from the previous with exactly one space or tab (any other spaces or tabs are treated as a literal part of the previous criterion). The ahk_id criterion can be combined with other criteria to test a window's title, class or other attributes: MouseGetPos ,, &id if WinExist("ahk_class Notepad ahk_id " id) MsgBox "The mouse is over Notepad." Last Found Window This is the window most recently found by WinExist, WinActive, WinWait[Not]Active, or WinWait. It can make scripts easier to create and maintain since WinTitle and WinText of the target window do not need to be repeated for every windowing function. In addition, scripts perform better because they don't need to search for the target window again after it has been found the first time. The last found window can be used by all of the windowing functions except WinWait, WinActivateBottom, GroupAdd, WinGetCount, and WinGetList. To use it, simply omit all four window parameters (WinTitle, WinText, ExcludeTitle, and ExcludeText). Each thread retains its own value of the last found window, meaning that if the current thread is interrupted by another, when the original thread is resumed it will still have its original value of the last found window, not that of the interrupting thread. If the last found window is a hidden Gui window, it can be used even when DetectHiddenWindows is turned off. This is often used in conjunction with the GUI's +LastFound option. The following example opens Notepad, waits until it exists and activates it: Run "Notepad" WinWait "Untitled - Notepad" WinActivate ; Use the window found by WinWait. The following example activates and maximizes the Notepad window found by WinExist: if WinExist("Untitled - Notepad") { WinActivate ; Use the window found by WinExist. WinMaximize ; Same as above. Send "Some text.{Enter}" } The following example activates the calculator found by WinExist and moves it to a new position: if not WinExist("Calculator") { ; ... } else { WinActivate ; Use the window found by WinExist. WinMove 40, 40 ; Same as above. } ï»¿; Context Sensitive Help in Any Editor (based on the v1 script by Rajat) ; https://www.autohotkey.com ; This script makes Ctrl+2 (or another hotkey of your choice) show the help file ; page for the selected AutoHotkey function or keyword. If nothing is selected, ; the function name will be extracted from the beginning of the current line. ; The hotkey below uses the clipboard to provide compatibility with the maximum ; number of editors (since ControlGet doesn't work with most advanced editors). ; It restores the original clipboard contents afterward, but as plain text, ; which seems better than nothing. $^2:: { ; The following values are in effect only for the duration of this hotkey thread. ; Therefore, there is no need to change them back to their original values ; because that is done automatically when the thread ends: SetWinDelay 10 SetKeyDelay 0 C_ClipboardPrev := A_Clipboard A_Clipboard := "" ; Use the highlighted word if there is one (since sometimes the user might ; intentionally highlight something that isn't a function): Send "^c" if !ClipWait(0.1) { ; Get the entire line because editors treat cursor navigation keys differently: Send "{home}+{end}^c" if !ClipWait(0.2) ; Rare, so no error is reported. { A_Clipboard := C_ClipboardPrev return } } C_Cmd := Trim(A_Clipboard) ; This will trim leading and trailing tabs & spaces. A_Clipboard := C_ClipboardPrev ; Restore the original clipboard for the user. Loop Parse, C_Cmd, "`s" ; The first space is the end of the function. { C_Cmd := A_LoopField break ; i.e. we only need one interation. } if !WinExist("AutoHotkey Help") { ; Determine AutoHotkey's location: try ahk_dir := RegRead

*("HKEY_LOCAL_MACHINE\SOFTWARE\AutoHotkey", "InstallDir") catch ; Not found, so look for it in some other common locations. { if A_AhkPath SplitPath A_AhkPath,, &ahk_dir else if FileExist("..\..\AutoHotkey.chm") ahk_dir := "..\.." else if FileExist(A_ProgramFiles "\AutoHotkey\AutoHotkey.chm") ahk_dir := A_ProgramFiles "\AutoHotkey" else { MsgBox "Could not find the AutoHotkey folder." return } } Run ahk_dir "\AutoHotkey.chm" WinWait "AutoHotkey Help" } ; The above has set the "last found" window which we use below: WinActivate WinWaitActive C_Cmd := StrReplace(C_Cmd, "#", "{#}") Send "!n{home}+{end}" C_Cmd "{enter}" } ï»¿; Easy Window Dragging ; https://www.autohotkey.com ; Normally, a window can only be dragged by clicking on its title bar. ; This script extends that so that any point inside a window can be dragged. ; To activate this mode, hold down CapsLock or the middle mouse button while ; clicking, then drag the window to a new position. ; Note: You can optionally release CapsLock or the middle mouse button after ; pressing down the mouse button rather than holding it down the whole time. ~MButton & LButton:: CapsLock & LButton:: EWD_MoveWindow(*) { CoordMode "Mouse" ; Switch to screen/absolute coordinates. MouseGetPos &EWD_MouseStartX, &EWD_MouseStartY, &EWD_MouseWin WinGetPos &EWD_OriginalPosX, &EWD_OriginalPosY,,, EWD_MouseWin if !WinGetMinMax (EWD_MouseWin) ; Only if the window isn't maximized SetTimer EWD_WatchMouse, 10 ; Track the mouse as the user drags it. EWD_WatchMouse() { if !GetKeyState ("LButton", "P") ; Button has been released, so drag is complete. { SetTimer , 0 return } if GetKeyState("Escape", "P") ; Escape has been pressed, so drag is cancelled. { SetTimer , 0 WinMove EWD_OriginalPosX, EWD_OriginalPosY,,, EWD_MouseWin return } ; Otherwise, reposition the window to match the change in mouse coordinates ; caused by the user having dragged the mouse: CoordMode "Mouse" MouseGetPos &EWD_MouseX, &EWD_MouseY WinGetPos &EWD_WinX, &EWD_WinY,,, EWD_MouseWin SetWinDelay -1 ; Makes the below move faster/smoother. WinMove EWD_WinX + EWD_MouseX - EWD_MouseStartX, EWD_WinY + EWD_MouseY - EWD_MouseStartY,,, EWD_MouseWin EWD_MouseStartX := EWD_MouseX ; Update for the next timer-call to this subroutine. EWD_MouseStartY := EWD_MouseY } } ï»¿; Easy Window Dragging -- KDE style (based on the v1 script by Jonny) ; https://www.autohotkey.com ; This script makes it much easier to move or resize a window: 1) Hold down ; the ALT key and LEFT-click anywhere inside a window to drag it to a new ; location; 2) Hold down ALT and RIGHT-click-drag anywhere inside a window ; to easily resize it; 3) Press ALT twice, but before releasing it the second ; time, left-click to minimize the window under the mouse cursor, right-click ; to maximize it, or middle-click to close it. ; The Double-Alt modifier is activated by pressing ; Alt twice, much like a double-click. Hold the second ; press down until you click. ; ; The shortcuts: ; Alt + Left Button : Drag to move a window. ; Alt + Right Button : Drag to resize a window. ; Double-Alt + Left Button : Minimize a window. ; Double-Alt + Right Button : Maximize/Restore a window. ; Double-Alt + Middle Button : Close a window. ; ; You can optionally release Alt after the first ; click rather than holding it down the whole time. ; This is the setting that runs smoothest on my ; system. Depending on your video card and cpu ; power, you may want to raise or lower this value. SetWinDelay 2 CoordMode "Mouse" g_DoubleAlt := false !LButton:: { global g_DoubleAlt ; Declare it since this hotkey function must modify it. if g_DoubleAlt { MouseGetPos ,, &KDE_id ; This message is mostly equivalent to WinMinimize, ; but it avoids a bug with PSPad. PostMessage 0x0112, 0xf020,,, KDE_id g_DoubleAlt := false return } ; Get the initial mouse position and window id, and ; abort if the window is maximized. MouseGetPos &KDE_X1, &KDE_Y1, &KDE_id if WinGetMinMax(KDE_id) return ; Get the initial window position. WinGetPos &KDE_WinX1, &KDE_WinY1,,, KDE_id Loop { if !GetKeyState("LButton", "P") ; Break if button has been released. break MouseGetPos &KDE_X2, &KDE_Y2 ; Get the current mouse position. KDE_X2 -= KDE_X1 ; Obtain an offset from the initial mouse position. KDE_Y2 -= KDE_Y1 KDE_WinX2 := (KDE_WinX1 + KDE_X2) ; Apply this offset to the window position. KDE_WinY2 := (KDE_WinY1 + KDE_Y2) WinMove KDE_WinX2, KDE_WinY2,,, KDE_id ; Move the window to the new position. } } !RButton:: { global g_DoubleAlt if g_DoubleAlt { MouseGetPos ,, &KDE_id ; Toggle between maximized and restored state. if WinGetMinMax(KDE_id) WinRestore KDE_id Else WinMaximize KDE_id g_DoubleAlt := false return } ; Get the initial mouse position and window id, and ; abort if the window is maximized. MouseGetPos &KDE_X1, &KDE_Y1, &KDE_id if WinGetMinMax(KDE_id) return ; Get the initial window position and size. WinGetPos &KDE_WinX1, &KDE_WinY1, &KDE_WinW, &KDE_WinH, KDE_id ; Define the window region the mouse is currently in. ; The four regions are Up and Left, Up and Right, Down and Left, Down and Right. if (KDE_X1 < KDE_WinX1 + KDE_WinW / 2) KDE_WinLeft := 1 else KDE_WinLeft := -1 if (KDE_Y1 < KDE_WinY1 + KDE_WinH / 2) KDE_WinUp := 1 else KDE_WinUp := -1 Loop { if !GetKeyState("RButton", "P") ; Break if button has been released. break MouseGetPos &KDE_X2, &KDE_Y2 ; Get the current mouse position. ; Get the current window position and size. WinGetPos &KDE_WinX1, &KDE_WinY1, &KDE_WinW, &KDE_WinH, KDE_id KDE_X2 -= KDE_X1 ; Obtain an offset from the initial mouse position. KDE_Y2 -= KDE_Y1 ; Then, act according to the defined region. WinMove KDE_WinX1 + (KDE_WinLeft+1)/2*KDE_X2 ; X of resized window , KDE_WinY1 + (KDE_WinUp+1)/2*KDE_Y2 ; Y of resized window , KDE_WinW - KDE_WinLeft *KDE_X2 ; W of resized window , KDE_WinH - KDE_WinUp *KDE_Y2 ; H of resized window , KDE_id KDE_X1 := (KDE_X2 + KDE_X1) ; Reset the initial position for the next iteration. KDE_Y1 := (KDE_Y2 + KDE_Y1) } } ; "Alt + MButton" may be simpler, but I like an extra measure of security for ; an operation like this. !MButton:: { global g_DoubleAlt if g_DoubleAlt { MouseGetPos ,, &KDE_id WinClose KDE_id g_DoubleAlt := false return } } ; This detects "double-clicks" of the alt key. ~Alt:: { global g_DoubleAlt := (A_PriorHotkey = "~Alt" and A_TimeSincePriorHotkey < 400) Sleep 0 KeyWait "Alt" ; This prevents the keyboard's auto-repeat feature from interfering. } ï»¿; HTML Entities Encoding ; https://www.autohotkey.com ; Similar to the Transform's HTML sub-command, this function converts a ; string into its HTML equivalent by translating characters whose ASCII ; values are above 127 to their HTML names (e.g. Â£ becomes £). In ; addition, the four characters "&<> are translated to "&<>. ; Finally, each linefeed (`n) is translated to `n (i.e. followed ; by a linefeed). ; In addition of the functionality above, Flags can be zero or a ; combination (sum) of the following values. If omitted, it defaults to 1. ; - 1: Converts certain characters to named expressions. e.g. â,¬ is ; converted to ; - 2: Converts certain characters to numbered expressions. e.g. â,¬ is ; converted to ; Only non-ASCII characters are affected. If Flags is the number 3, ; numbered expressions are used only where a named expression is not ; available. The following characters are always converted: <>"& and `n ; (line feed). EncodeHTML(String, Flags := 1) { static TRANS_HTML_NAMED := 1 static TRANS_HTML_NUMBERED := 2 static ansi := ["euro", "#129", "sbquo", "fnof", "bdquo", "hellip", "dagger", "Dagger", "circ", "permil", "Scaron", "lsaquo", "OElig", "#141", "#381", "#143", "#144", "lsquo", "rsquo", "ldquo", "rdquo", "bull", "ndash", "mdash", "tilde", "trade", "scaron", "rsaquo", "oelig", "#157", "#382", "Yuml", "nbsp", "iexcl", "cent", "pound", "curren", "yen", "brvbar", "sect", "uml", "copy", "ordf", "laquo", "not", "shy", "reg", "macr", "deg", "plusmn", "sup2", "sup3", "acute", "micro", "para", "middot", "cedil", "sup1", "ordm", "raquo", "frac14", "frac12", "frac34", "iquest", "Agrave", "Aacute", "Acirc", "Atilde", "Auml", "Aring", "AElig", "Ccedil", "Egrave", "Eacute", "Ecirc", "Euml", "Igrave", "Iacute", "Icirc", "Iuml", "ETH", "Ntilde", "Ograve", "Oacute", "Ocirc", "Otilde", "Ouml", "times", "Oslash", "Ugrave", "Uacute", "Ucirc", "Uuml", "Yacute", "THORN", "szlig", "agrave", "aacute", "acirc", "atilde", "auml", "aring", "aelig", "ccedil", "egrave", "eacute", "ecirc", "euml", "igrave", "iacute", "icirc", "iuml", "eth", "ntilde", "ograve", "oacute", "ocirc", "otilde", "ouml", "divide", "oslash", "ugrave", "uacute", "ucirc", "uuml", "yacute", "thorn", "yuml"] static unicode := {0x20AC:1, 0x201A:3, 0x0192:4, 0x201E:5, 0x2026:6, 0x2020:7, 0x2021:8, 0x02C6:9, 0x2030:10, 0x0160:11, 0x2039:12, 0x0152:13, 0x2018:18, 0x2019:19, 0x201C:20, 0x201D:21, 0x2022:22, 0x2013:23, 0x2014:24, 0x02DC:25, 0x2122:26, 0x0161:27, 0x203A:28, 0x0153:29, 0x0178:32} out := "" for i, char in StrSplit(String) { code := Ord(char) switch code { case 10: out .= "`n" case 34: out .= """ case 38: out .= "&" case 60: out .= "<" case 62: out .= ">" default: if (code >= 160 && code <= 255) { if (Flags & TRANS_HTML_NAMED) out .= "&" ansi[code-127] ";" else if (Flags & TRANS_HTML_NUMBERED) out .= "&#" code ";" else out .= char } else if (code > 255) { if (Flags & TRANS_HTML_NAMED && unicode.HasOwnProp(code)) out .= "&" ansi[unicode.%code%] ";" else if (Flags & TRANS_HTML_NUMBERED) out .= "&#" code ";" else out .= char } else { if (code >= 128 && code <= 159) out .= "&" ansi[code-127] ";" else out .= char } } } return out }ï»¿; Easy Access to Favorite Folders (based on the v1 script by Savage) ; https://www.autohotkey.com ; When you click the middle mouse button while certain types of ; windows are active, this script displays a menu of your favorite ; folders. Upon selecting a favorite, the script will instantly ; switch to that folder within the active window. The following ; window types are supported: 1) Standard file-open or file-save ; dialogs; 2) Explorer windows; 3) Console (command prompt) windows. ; The menu can also be optionally shown for unsupported window ; types, in which case the chosen favorite will be opened as a new ; Explorer window. ; CONFIG: CHOOSE YOUR HOTKEY ; If your mouse has more than 3 buttons, you could try using ; XButton1 (the 4th) or XButton2 (the 5th) instead of MButton. ; You could also use a modified mouse button (such as ^MButton) or ; a keyboard hotkey. In the case of MButton, the tilde (~) prefix ; is used so that MButton's normal functionality is not lost when ; you click in other window types, such as a browser. The presence ; of a tilde tells the script to avoid showing the menu for ; unsupported window types. In other words, if there is no tilde, ; the hotkey will always display the menu; and upon selecting a ; favorite while an unsupported window type is active, a new ; Explorer window will be opened to display the contents of that ; folder. g_Hotkey := "~MButton" ; CONFIG: CHOOSE YOUR FAVORITES ; Update the special commented section below to list your favorite ; folders. Specify the name of the menu item first, followed by a ; semicolon, followed by the name of the actual path of the favorite. ; Use a blank line to create a separator line. /* ITEMS IN FAVORITES MENU <-- Do not change this string. Desktop ; %USERPROFILE%\Desktop Favorites ; %USERPROFILE%\Favorites Documents ; %USERPROFILE%\Documents Program Files; %PROGRAMFILES% */ ; END OF CONFIGURATION SECTION ; Do not make changes below this point unless you want to change ; the basic functionality of the script. #SingleInstance ; Needed since the hotkey is dynamically created. g_AlwaysShowMenu := true g_Paths := [] g_Menu := Menu() g_window_id := 0 g_class := "" Hotkey g_Hotkey, DisplayMenu if SubStr(g_Hotkey, 1, 1) = "~" ; Show menu only for certain window types. g_AlwaysShowMenu := false if A_IsCompiled ; Read the menu items from an external file. FavoritesFile := A_ScriptDir "\Favorites.ini" else ; Read the menu items directly from this script file. FavoritesFile := A_ScriptFullPath ;----Read the configuration file. AtStartingPos := false FileExt := "" Loop Read, FavoritesFile { if FileExt != "Exe" { ; Since the menu items are being read directly from this ; script, skip over all lines until the starting line is ; arrived at. if !AtStartingPos { if InStr (A_LoopReadLine, "ITEMS IN FAVORITES MENU") AtStartingPos := true continue ; Start a new loop iteration. } ; Otherwise, the closing comment symbol marks the end of the list. if A_LoopReadLine = "*/" break ; terminate the loop } if !A_LoopReadLine ; Blank indicates a separator line. { ; Menu separator lines must also be pushed to the array ; to be compatible with ItemPos: g_Paths.Push("") g_Menu.Add() } else { line := StrSplit(A_LoopReadLine, ";", "`s`t") ; Resolve any references to variables within either field, and ; create a new array element containing the path of this favorite: g_Paths.Push(line[2]) g_Menu.Add(line[1], OpenFavorite) } } ;---- Open the selected favorite OpenFavorite(ItemName, ItemPos, *) { ; Fetch the array element that corresponds to the selected menu item: path := g_Paths[ItemPos] if path = "" return if g_class = "#32770" ; It's a dialog. { ; Activate the window so that if the user is middle-clicking ; outside the dialog, subsequent clicks will also work:*

WinActivate g_window_id ; Retrieve any filename that might already be in the field so ; that it can be restored after the switch to the new folder: text := ControlGetText ("Edit1", g_window_id) ControlSetText path, "Edit1", g_window_id ControlFocus "Edit1", g_window_id ControlSend "{Enter}", "Edit1", g_window_id Sleep 100 ; It needs extra time on some dialogs or in some cases. ControlSetText text, "Edit1", g_window_id return } else if g_class = "CabinetWClass" ; In Explorer, switch folders. { ControlClick "ToolbarWindow323", g_window_id,,,, "NA x1 y1" ControlFocus "Edit1", g_window_id ControlSetText path, "Edit1", g_window_id ; Tekl reported the following: "If I want to change to Folder L:\folder ; then the addressbar shows http://www.L:\folder.com. To solve this, ; I added a {right} before {Enter}": ControlSend "{Right}{Enter}", "Edit1", g_window_id return } else if g_class = "ConsoleWindowClass" ; In a console window, CD to that directory { WinActivate g_window_id ; Because sometimes the mclick deactivates it. SetKeyDelay 0 ; This will be in effect only for the duration of this thread. if InStr(path, ":") ; It contains a drive letter { path_drive := SubStr(path, 1, 1) Send path_drive ":{enter}" } Send "cd " path "{Enter}" return } ; Since the above didn't return, one of the following is true: ; 1) It's an unsupported window type but g_AlwaysShowMenu is true. Run "explorer " path ; Might work on more systems without double quotes. } ;----Display the menu DisplayMenu(*) { ; These first few variables are set here and used by OpenFavorite: try global g_window_id := WinGetID("A") try global g_class := WinGetClass (g_window_id) if g_AlwaysShowMenu = false ; The menu should be shown only selectively. { if !(g_class ~= "#32770|ExploreWClass|CabinetWClass|ConsoleWindowClass") return ; Since it's some other window type, don't display menu. } ; Otherwise, the menu should be presented for this type of window: g_Menu.Show() }

AutoHotkey Script Showcase

This showcase lists some scripts created by different authors which show what AutoHotkey might be capable of. For more ready-to-run scripts and functions, see AutoHotkey v2 Scripts and Functions Forum.

Table of Contents

Context Sensitive Help in Any Editor

Based on the v1 script by Rajat

This script makes `Ctrl+2` (or another hotkey of your choice) show the help file page for the selected AutoHotkey function or keyword. If nothing is selected, the function name will be extracted from the beginning of the current line.

Show code

Easy Window Dragging

Normally, a window can only be dragged by clicking on its title bar. This script extends that so that any point inside a window can be dragged. To activate this mode, hold down `CapsLock` or the middle mouse button while clicking, then drag the window to a new position.

Show code

Easy Window Dragging (KDE style)

Based on the v1 script by Jonny

This script makes it much easier to move or resize a window: 1) Hold down `Alt` and LEFT-click anywhere inside a window to drag it to a new location; 2) Hold down `Alt` and RIGHT-click-drag anywhere inside a window to easily resize it; 3) Press `Alt` twice, but before releasing it the second time, left-click to minimize the window under the mouse cursor, right-click to maximize it, or middle-click to close it.

Show code

Easy Access to Favorite Folders

Based on the v1 script by Savage

When you click the middle mouse button while certain types of windows are active, this script displays a menu of your favorite folders. Upon selecting a favorite, the script will instantly switch to that folder within the active window. The following window types are supported: 1) Standard file-open or file-save dialogs; 2) Explorer windows; 3) Console (command prompt) windows. The menu can also be optionally shown for unsupported window types, in which case the chosen favorite will be opened as a new Explorer window.

Show code

IntelliSense

Based on the v1 script by Rajat

This script watches while you edit an AutoHotkey script. When it sees you type a command followed by a comma or space, it displays that command's parameter list to guide you. In addition, you can press `Ctrl+F1` (or another hotkey of your choice) to display that command's page in the help file. To dismiss the parameter list, press `Esc` or `Enter`.

Show code

Using a Joystick as a Mouse

This script converts a joystick into a three-button mouse. It allows each button to drag just like a mouse button and it uses virtually no CPU time. Also, it will move the cursor faster depending on how far you push the joystick from center. You can personalize various settings at the top of the script.

Show code

Joystick Test Script

This script helps determine the button numbers and other attributes of your joystick. It might also reveal if your joystick is in need of calibration; that is, whether the range of motion of each of its axes is from 0 to 100 percent as it should be. If calibration is needed, use the operating system's control panel or the software that came with your joystick.

Show code

On-Screen Keyboard

Based on the v1 script by Jon

This script creates a mock keyboard at the bottom of your screen that shows the keys you are pressing in real time. I made it to help me to learn to touch-type (to get used to not looking at the keyboard). The size of the on-screen keyboard can be customized at the top of the script. Also, you can double-click the tray icon to show or hide the keyboard.

Show code

Minimize Window to Tray Menu

This script assigns a hotkey of your choice to hide any window so that it becomes an entry at the bottom of the script's tray menu. Hidden windows can then be unhidden individually or all at once by selecting the corresponding item on the menu. If the script exits for any reason, all the windows that it hid will be unhidden automatically.

Show code

Changing MsgBox's Button Names

This is a working example script that uses a timer to change the names of the buttons in a message box. Although the button names are changed, the MsgBox's return value still requires that the buttons be referred to by their original names.

Show code

Numpad 000 Key

This example script makes the special `000` that appears on certain keypads into an equals key. You can change the action by replacing the `Send "="` line with line(s) of your choice.

Show code

Using Keyboard Numpad as a Mouse

Based on the v1 script by deguix

This script makes mousing with your keyboard almost as easy as using a real mouse (maybe even easier for some tasks). It supports up to five mouse buttons and the turning of the mouse wheel. It also features customizable movement speed, acceleration, and "axis inversion".

Show code

Seek (Search the Start Menu)

Based on the v1 script by Phi

Navigating the Start Menu can be a hassle, especially if you have installed many programs over time. 'Seek' lets you specify a case-insensitive key word/phrase that it will use to filter only the matching programs and directories from the Start Menu, so that you can easily open your target program from a handful of matched entries. This eliminates the drudgery of searching and traversing the Start Menu.

Show code

ToolTip Mouse Menu

Based on the v1 script by Rajat

This script displays a popup menu in response to briefly holding down the middle mouse button. Select a menu item by left-clicking it. Cancel the menu by left-clicking outside of it. A recent improvement is that the contents of the menu can change depending on which type of window is active (Notepad and Word are used as examples

here).

Show code

Volume On-Screen-Display (OSD)

Based on the v1 script by Rajat

This script assigns hotkeys of your choice to raise and lower the master volume.

Show code

Window Shading

Based on the v1 script by Rajat

This script reduces a window to its title bar and then back to its original size by pressing a single hotkey. Any number of windows can be reduced in this fashion (the script remembers each). If the script exits for any reason, all "rolled up" windows will be automatically restored to their original heights.

Show code

WinLIRC Client

This script receives notifications from WinLIRC whenever you press a button on your remote control. It can be used to automate Winamp, Windows Media Player, etc. It's easy to configure. For example, if WinLIRC recognizes a button named "VolUp" on your remote control, create a label named VolUp and beneath it use the function `SoundSetVolume "+5"` to increase the soundcard's volume by 5%.

Show code

HTML Entities Encoding

***Similar to AutoHotkey v1's Transform HTML, this function converts a string into its HTML equivalent by translating characters whose ASCII values are above 127 to their HTML names (e.g. £ becomes `&pound;`). In addition, the four characters `"&<>` are translated to `&quot;&<>`. Finally, each linefeed (`n) is translated to `
`n (i.e.
 followed by a linefeed).***

Show code

AutoHotkey v1 Scripts and Functions Forum

This forum contains many more scripts, but most scripts will not run as-is on AutoHotkey v2.0.

AutoHotkey v1 Scripts and Functions Forum

ï»¿; IntelliSense (based on the v1 script by Rajat) ; https://www.autohotkey.com ; This script watches while you edit an AutoHotkey script. When it sees you ; type a command followed by a comma or space, it displays that command's ; parameter list to guide you. In addition, you can press Ctrl+F1 (or ; another hotkey of your choice) to display that command's page in the help ; file. To dismiss the parameter list, press Escape or Enter. ; CONFIGURATION SECTION: Customize the script with the following variables. ; The hotkey below is pressed to display the current command's page in the ; help file: g_HelpHotkey := "^F1" ; The string below must exist somewhere in the active window's title for ; IntelliSense to be in effect while you're typing. Make it blank to have ; IntelliSense operate in all windows. Make it Pad to have it operate in ; editors such as Metapad, Notepad, and Textpad. Make it .ahk to have it ; operate only when a .ahk file is open in Notepad, Metapad, etc. g_Editor := ".ahk" ; If you wish to have a different icon for this script to distinguish it from ; other scripts in the tray, provide the filename below (leave blank to have ; no icon). For example: E:\stuff\Pics\icons\GeoIcons\Information.ico g_Icon := "" ; END OF CONFIGURATION SECTION (do not make changes below this point unless ; you want to change the basic functionality of the script). SetKeyDelay 0 #SingleInstance g_ThisCmd := "" g_HelpOn := "" g_Cmds := [] g_FullCmds := [] g_Word := "" if g_HelpHotkey != "" Hotkey g_HelpHotkey, HelpHotkey ; Change tray icon (if one was specified in the configuration section above): if g_Icon != "" if FileExist(g_Icon) TraySetIcon g_Icon ; Determine AutoHotkey's location: try ahk_dir := RegRead("HKEY_LOCAL_MACHINE\SOFTWARE\AutoHotkey", "InstallDir") catch ; Not found, so look for it in some other common locations. { if A_AhkPath SplitPath A_AhkPath,, &ahk_dir else if FileExist("..\..\AutoHotkey.chm") ahk_dir := "..\.." else if FileExist(A_ProgramFiles "\AutoHotkey\AutoHotkey.chm") ahk_dir := A_ProgramFiles "\AutoHotkey" else { MsgBox "Could not find the AutoHotkey folder." ExitApp } } g_AhkHelpFile := ahk_dir "\AutoHotkey.chm" ; Read command syntaxes; can be found in AHK Basic, but it's outdated: Loop Read, ahk_dir "\Extras\Editors\Syntax\Commands.txt" { FullCmd := A_LoopReadLine ; Directives have a first space instead of a first comma. ; So use whichever comes first as the end of the command name: cPos := InStr(FullCmd, "(") sPos := InStr(FullCmd, "`s") if (!cPos or (cPos > sPos and sPos)) EndPos := sPos else EndPos := cPos if EndPos CurrCmd := SubStr(FullCmd, 1, EndPos - 1) else ; This is a directive/command with no parameters. CurrCmd := A_LoopReadLine CurrCmd := StrReplace(CurrCmd, "[") CurrCmd := StrReplace(CurrCmd, "`s") FullCmd := StrReplace(FullCmd, "``n", "`n") FullCmd := StrReplace(FullCmd, "``t", "`t") ; Make arrays of command names and full cmd syntaxes: g_Cmds.Push(CurrCmd) g_FullCmds.Push(FullCmd) } ; Use the Input function to watch for commands that the user types: Loop { ; Editor window check: if !WinActive(g_Editor) { ToolTip Sleep 500 Continue } ; Get all keys till endkey: Hook := Input("V", "{Enter}{Escape}{Space},") g_Word := Hook.Input EndKey := Hook.EndKey ; ToolTip is hidden in these cases: if EndKey = "Enter" or EndKey = "Escape" { ToolTip Continue } ; Editor window check again! if !WinActive(g_Editor) { ToolTip Continue } ; Compensate for any indentation that is present: g_Word := StrReplace(g_Word, "`s") g_Word := StrReplace(g_Word, "`t") if g_Word = "" Continue ; Check for commented line: Check := SubStr(g_Word, 1, 1) if (Check = ";" or g_Word = "If") ; "If" seems a little too annoying to show tooltip for. Continue ; Match word with command: Index := "" for Cmd in g_Cmds { ; The value put into g_ThisCmd is also used by the HelpHotkey function: g_ThisCmd := Cmd if (g_Word = g_ThisCmd) { Index := A_Index g_HelpOn := g_ThisCmd break } } ; If no match then resume watching user input: if Index = "" Continue ; Show matched command to guide the user: ThisFullCmd := g_FullCmds[Index] CaretGetPos &CaretX, &CaretY ToolTip ThisFullCmd, CaretX, CaretY + 20 } ; This script was originally written for AutoHotkey v1. ; Input() is a rough reproduction of the Input command. Input(Options:="", EndKeys:="", MatchList:="") { static ih if IsSet(&ih) && ih.InProgress ih.Stop() ih := InputHook(Options, EndKeys, MatchList) ih.Start() ih.Wait() return ih } HelpHotkey(*) { global g_ThisCmd ; Declared because this function modifies it. if !WinActive(g_Editor) return ToolTip ; Turn off syntax helper since there is no need for it now. SetTitleMatchMode 1 ; In case it's 3. This setting is in effect only for this thread. if !WinExist("AutoHotkey Help") { if !FileExist(g_AhkHelpFile) { MsgBox "Could not find the help file: " g_AhkHelpFile return } Run g_AhkHelpFile WinWait "AutoHotkey Help" } if g_ThisCmd = "" ; Instead, use what was most recently typed. g_ThisCmd := g_Word ; The above has set the "last found" window which we use below: WinActivate WinWaitActive g_ThisCmd := StrReplace(g_ThisCmd, "#", "{#}") ; Replace leading #, if any. Send "!n{home}+{end}" g_HelpOn "{enter}" } ï»¿; Using a Joystick as a Mouse ; https://www.autohotkey.com ; This script converts a joystick into a three-button mouse. It allows each ; button to drag just like a mouse button and it uses virtually no CPU time. ; Also, it will move the cursor faster depending on how far you push the joystick ; from center. You can personalize various settings at the top of the script. ; Increase the following value to make the mouse cursor move faster: JoyMultiplier := 0.30 ; Decrease the following value to require less joystick displacement-from-center ; to start moving the mouse. However, you may need to calibrate your joystick ; -- ensuring it's properly centered -- to avoid cursor drift. A perfectly tight ; and centered joystick could use a value of 1: JoyThreshold := 3 ; Change the following to true to invert the Y-axis, which causes the mouse to ; move vertically in the direction opposite the stick: InvertYAxis := false ; Change these values to use joystick button numbers other than 1, 2, and 3 for ; the left, right, and middle mouse buttons. Available numbers are 1 through 32. ; Use the Joystick Test Script to find out your

*joystick's numbers more easily. ButtonLeft := 1 ButtonRight := 2 ButtonMiddle := 3 ; If your joystick has a POV control, you can use it as a mouse wheel. The ; following value is the number of milliseconds between turns of the wheel. ; Decrease it to have the wheel turn faster: WheelDelay := 250 ; If your system has more than one joystick, increase this value to use a joystick ; other than the first: JoystickNumber := 1 ; END OF CONFIG SECTION -- Don't change anything below this point unless you want ; to alter the basic nature of the script. #SingleInstance JoystickPrefix := JoystickNumber "Joy" Hotkey JoystickPrefix . ButtonLeft, ClickButtonLeft Hotkey JoystickPrefix . ButtonRight, ClickButtonRight Hotkey JoystickPrefix . ButtonMiddle, ClickButtonMiddle ; Calculate the axis displacements that are needed to start moving the cursor: JoyThresholdUpper := 50 + JoyThreshold JoyThresholdLower := 50 - JoyThreshold if InvertYAxis YAxisMultiplier := -1 else YAxisMultiplier := 1 SetTimer WatchJoystick, 10 ; Monitor the movement of the joystick. JoyInfo := GetKeyState(JoystickNumber "JoyInfo") if InStr(JoyInfo, "P") ; Joystick has POV control, so use it as a mouse wheel. SetTimer MouseWheel, WheelDelay ; The functions below do not use KeyWait because that would sometimes trap the ; WatchJoystick quasi-thread beneath the wait-for-button-up thread, which would ; effectively prevent mouse-dragging with the joystick. ClickButtonLeft(*) { SetMouseDelay -1 ; Makes movement smoother. MouseClick "Left",,, 1, 0, "D" ; Hold down the left mouse button. SetTimer WaitForLeftButtonUp, 10 WaitForLeftButtonUp() { if GetKeyState(A_ThisHotkey) return ; The button is still, down, so keep waiting. ; Otherwise, the button has been released. SetTimer , 0 SetMouseDelay -1 ; Makes movement smoother. MouseClick "Left",,, 1, 0, "U" ; Release the mouse button. } } ClickButtonRight(*) { SetMouseDelay -1 ; Makes movement smoother. MouseClick "Right",,, 1, 0, "D" ; Hold down the right mouse button. SetTimer WaitForRightButtonUp, 10 WaitForRightButtonUp() { if GetKeyState(A_ThisHotkey) return ; The button is still, down, so keep waiting. ; Otherwise, the button has been released. SetTimer , 0 MouseClick "Right",,, 1, 0, "U" ; Release the mouse button. } } ClickButtonMiddle(*) { SetMouseDelay -1 ; Makes movement smoother. MouseClick "Middle",,, 1, 0, "D" ; Hold down the right mouse button. SetTimer WaitForMiddleButtonUp, 10 WaitForMiddleButtonUp() { if GetKeyState(A_ThisHotkey) return ; The button is still, down, so keep waiting. ; Otherwise, the button has been released. SetTimer , 0 MouseClick "Middle",,, 1, 0, "U" ; Release the mouse button. } } WatchJoystick() { global MouseNeedsToBeMoved := false ; Set default. joyx := GetKeyState(JoystickNumber "JoyX") joyy := GetKeyState(JoystickNumber "JoyY") if joyx > JoyThresholdUpper { MouseNeedsToBeMoved := true DeltaX := Round(joyx - JoyThresholdUpper) } else if joyx < JoyThresholdLower { MouseNeedsToBeMoved := true DeltaX := Round(joyx - JoyThresholdLower) } else DeltaX := 0 if joyy > JoyThresholdUpper { MouseNeedsToBeMoved := true DeltaY := Round(joyy - JoyThresholdUpper) } else if joyy < JoyThresholdLower { MouseNeedsToBeMoved := true DeltaY := Round(joyy - JoyThresholdLower) } else DeltaY := 0 if MouseNeedsToBeMoved { SetMouseDelay -1 ; Makes movement smoother. MouseMove DeltaX * JoyMultiplier, DeltaY * JoyMultiplier * YAxisMultiplier, 0, "R" } } MouseWheel() { global JoyPOV := GetKeyState(JoystickNumber "JoyPOV") if JoyPOV = -1 ; No angle. return if (JoyPOV > 31500 or JoyPOV < 4500) ; Forward Send "{WheelUp}" else if JoyPOV >= 13500 and JoyPOV <= 22500 ; Back Send "{WheelDown}" } ï»¿; Joystick Test Script ; https://www.autohotkey.com ; This script helps determine the button numbers and other attributes ; of your joystick. It might also reveal if your joystick is in need ; of calibration; that is, whether the range of motion of each of its ; axes is from 0 to 100 percent as it should be. If calibration is ; needed, use the operating system's control panel or the software ; that came with your joystick. ; July 16, 2016: Revised code for AHK v2 compatibility ; July 6, 2005 : Added auto-detection of joystick number. ; May 8, 2005 : Fixed: JoyAxes is no longer queried as a means of ; detecting whether the joystick is connected. Some joysticks are ; gamepads and don't have even a single axis. ; If you want to unconditionally use a specific joystick number, change ; the following value from 0 to the number of the joystick (1-16). ; A value of 0 causes the joystick number to be auto-detected: JoystickNumber := 0 ; END OF CONFIG SECTION. Do not make changes below this point unless ; you wish to alter the basic functionality of the script. ; Auto-detect the joystick number if called for: if JoystickNumber <= 0 { Loop 16 ; Query each joystick number to find out which ones exist. { if GetKeyState(A_Index "JoyName") { JoystickNumber := A_Index break } } if JoystickNumber <= 0 { MsgBox "The system does not appear to have any joysticks." ExitApp } } #SingleInstance joy_buttons := GetKeyState(JoystickNumber "JoyButtons") joy_name := GetKeyState(JoystickNumber "JoyName") joy_info := GetKeyState(JoystickNumber "JoyInfo") Loop { buttons_down := "" Loop joy_buttons { if GetKeyState (JoystickNumber "Joy" A_Index) buttons_down .= " " A_Index } axis_info := "X" Round(GetKeyState(JoystickNumber "JoyX")) axis_info .= " Y" Round(GetKeyState (JoystickNumber "JoyY")) if InStr(joy_info, "Z") axis_info .= " Z" Round(GetKeyState(JoystickNumber "JoyZ")) if InStr(joy_info, "R") axis_info .= " R" Round (GetKeyState(JoystickNumber "JoyR")) if InStr(joy_info, "U") axis_info .= " U" Round(GetKeyState(JoystickNumber "JoyU")) if InStr(joy_info, "V") axis_info .= " V" Round(GetKeyState(JoystickNumber "JoyV")) if InStr(joy_info, "P") axis_info .= " POV" Round(GetKeyState(JoystickNumber "JoyPOV")) ToolTip (joy_name " (#" JoystickNumber "): " axis_info " Buttons Down: " buttons_down " (right-click the tray icon to exit)") Sleep 100 } return ï»¿; On-Screen Keyboard (based on the v1 script by Jon) ; https://www.autohotkey.com ; This script creates a mock keyboard at the bottom of your screen that shows ; the keys you are pressing in real time. I made it to help me to learn to ; touch-type (to get used to not looking at the keyboard). The size of the ; on-screen keyboard can be customized at the top of the script. Also, you ; can double-click the tray icon to show or hide the keyboard. ;---- Configuration Section: Customize the size of the on-screen keyboard and ; other options here. ; Changing this font size will make the entire on-screen keyboard get ; larger or smaller: k_FontSize := 10 k_FontName := "Verdana" ; This can be blank to use the system's default font. k_FontStyle := "Bold" ; Example of an alternative: Italic Underline ; Names for the tray menu items: k_MenuItemHide := "Hide on-screen &keyboard" k_MenuItemShow := "Show on-screen &keyboard" ; To have the keyboard appear on a monitor other than the primary, specify ; a number such as 2 for the following variable. Leave it unset to use ; the primary: k_Monitor := unset ;---- End of configuration section. Don't change anything below this point ; unless you want to alter the basic nature of the script. ;---- Create a Gui window for the on-screen keyboard: MyGui := Gui("-Caption +ToolWindow +AlwaysOnTop +Disabled") MyGui.SetFont("s" k_FontSize " " k_FontStyle, k_FontName) MyGui.MarginY := 0, MyGui.MarginX := 0 ;---- Alter the tray icon menu: A_TrayMenu.Delete A_TrayMenu.Add k_MenuItemHide, k_ShowHide A_TrayMenu.Add "&Exit", (*) => ExitApp() A_TrayMenu.Default := k_MenuItemHide ;---- Add a button for each key: ; The keyboard layout: k_Layout := [["``", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-", "=", "Backspace:3"], ["Tab:3", "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "[", "]", "\"], ["CapsLock:3", "A", "S", "D", "F", "G", "H", "J", "K", "L", ";", "'", "Enter:2"], ["LShift:3", "Z", "X", "C", "V", "B", "N", "M", ",", ".", "/", "Shift:3"], ["LCtrl:2", "LWin:2", "LAlt:2", "Space:2", "RAlt:2", "RWin:2", "AppsKey:2", "RCtrl:2"]] ; Traverse the keys of the keyboard layout: for n, k_Row in k_Layout for i, k_Key in k_Row { k_KeyWidthMultiplier := 1 ; Get custom key width multiplier: if RegExMatch(k_Key, "(.+):(\d)", &m) { k_Key := m[1] k_KeyWidthMultiplier := m[2] } ; Get localized key name: k_KeyNameText := GetKeyNameText(k_Key, 0, 1) ; Windows key names start with left or right so replace it: if (k_Key = "LWin" || k_Key = "RWin") k_KeyNameText := "Win" ; Truncate the key name: if (StrLen(k_Key) > 1) k_KeyNameText := Trim(SubStr (k_KeyNameText, 1, 5)) else k_KeyNameText := k_Key ; Convert to uppercase: k_KeyNameText := StrUpper(k_KeyNameText) ; Calculate object dimensions based on chosen font size: k_KeyHeight := k_FontSize * 3 opt := "h" k_KeyHeight " w" k_KeyHeight * k_KeyWidthMultiplier " -Wrap x+m" if (i = 1) opt .= " y+m xm" ; Add the button: Btn := MyGui.Add("Button", opt, k_KeyNameText) ; When a key is pressed by the user, click the corresponding button on-screen: Hotkey("~*" k_Key, k_KeyPress.bind(Btn)) } ;---- Position the keyboard at the bottom of the screen (taking into account ; the position of the taskbar): MyGui.Show("Hide") ; Required to get the window's calculated width and height. ; Calculate window's X-position: MonitorGetWorkArea(k_Monitor?, &WL,, &WR, &WB) MyGui.GetPos(,, &k_width, &k_height) k_xPos := (WR - WL - k_width) / 2 ; Calculate position to center it horizontally. ; The following is done in case the window will be on a non-primary monitor ; or if the taskbar is anchored on the left side of the screen: k_xPos += WL ; Calculate window's Y-position: k_yPos := WB - k_height ;---- Show the window: MyGui.Show("x" k_xPos " y" k_yPos " NA") ;---- Function definitions: k_KeyPress(BtnCtrl, *) { BtnCtrl.Opt("Default") ; Highlight the last pressed key. ControlClick(, BtnCtrl,,,, "D") KeyWait(SubStr(A_ThisHotkey, 3)) ControlClick(, BtnCtrl,,,, "U") } k_ShowHide(*) { static isVisible := true if isVisible { MyGui.Hide A_TrayMenu.Rename k_MenuItemHide, k_MenuItemShow isVisible := false } else { MyGui.Show A_TrayMenu.Rename k_MenuItemShow, k_MenuItemHide isVisible := true } } GetKeyNameText(Key, Extended := false, DoNotCare := false) { Params := (GetKeySC(Key) << 16) | (Extended << 24) | (DoNotCare << 25) KeyNameText := Buffer(64, 0) DllCall("User32.dll\GetKeyNameText", "Int", Params, "Ptr", KeyNameText, "Int", 32) return StrGet(KeyNameText) } ï»¿; Minimize Window to Tray Menu ; https://www.autohotkey.com ; This script assigns a hotkey of your choice to hide any window so that ; it becomes an entry at the bottom of the script's tray menu. Hidden ; windows can then be unhidden individually or all at once by selecting ; the corresponding item on the menu. If the script exits for any reason, ; all the windows that it hid will be unhidden automatically. ; CONFIGURATION SECTION: Change the below values as desired. ; This is the maximum number of windows to allow to be hidden (having a ; limit helps performance): g_MaxWindows := 50 ; This is the hotkey used to hide the active window: g_Hotkey := "#h" ; Win+H ; This is the hotkey used to unhide the last hidden window: g_UnHotkey := "#u" ; Win+U ; If you prefer to have the tray menu empty of all the standard items, ; such as Help and Pause, use False. Otherwise, use True: g_StandardMenu := false ; These next few performance settings help to keep the action within the ; A_HotkeyModifierTimeout period, and thus avoid the need to release and ; press down the hotkey's modifier if you want to hide more than one ; window in a row. These settings are not needed if you choose to have ; the script use the keyboard hook via InstallKeybdHook or other means: A_HotkeyModifierTimeout := 100 SetWinDelay 10 SetKeyDelay 0 #SingleInstance ; Allow only one instance of this script to be running. Persistent ; END OF CONFIGURATION SECTION (do not make changes below this point ; unless you want to change the basic functionality of the script). g_WindowIDs := [] g_WindowTitles := [] Hotkey g_Hotkey, Minimize Hotkey g_UnHotkey, UnMinimize ; If the user terminates the script by any means, unhide all the ; windows first: OnExit RestoreAllThenExit if g_StandardMenu = true A_TrayMenu.Add else { A_TrayMenu.Delete A_TrayMenu.Add "E&xit and Unhide All", RestoreAllThenExit } A_TrayMenu.Add "&Unhide All Hidden Windows", RestoreAll A_TrayMenu.Add ; Another separator line to make the above more special. g_MaxLength := 260 ; Reduce this to restrict the width of the menu. Minimize(*) { if g_WindowIDs.Length >= g_MaxWindows { MsgBox "No more than " g_MaxWindows " may be hidden simultaneously." return } ; Set the "last found window" to simplify and help performance. ; Since in certain cases it is possible for there to be no active window, ; a timeout has been added: if !WinWait("A",, 2) ; It timed out, so do nothing. return ; Otherwise, the "last found window" has been set and can now be used: ActiveID := WinGetID() ActiveTitle := WinGetTitle() ActiveClass := WinGetClass() if ActiveClass ~= "Shell_TrayWnd|Progman" { MsgBox "The desktop and taskbar cannot be hidden." return } ; Because hiding the window won't deactivate it, activate the window ; beneath this one (if any). I tried other ways, but they wound up ; activating the task bar. This way sends the active window (which is ; about to be hidden) to the back of the stack, which seems best: Send "!{esc}" ; Hide it only now that WinGetTitle/WinGetClass above have been run (since ; by default,*

```
those functions cannot detect hidden windows): WinHide ; If the title is blank, use the class instead. This serves two purposes: ; 1) A more meaningful name is used as the menu name. ; 2) Allows the menu item to be created (otherwise, blank items wouldn't ; be handled correctly by the various routines below). if ActiveTitle = "" ActiveTitle := "ahk_class " ActiveClass ; Ensure the title is short enough to fit. ActiveTitle also serves to ; uniquely identify this particular menu item. ActiveTitle := SubStr(ActiveTitle, 1, g_MaxLength) ; In addition to the tray menu requiring that each menu item name be ; unique, it must also be unique so that we can reliably look it up in ; the array when the window is later unhidden. So make it unique if it ; isn't already: for WindowTitle in g_WindowTitles { if WindowTitle = ActiveTitle { ; Match found, so it's not unique. ActiveIDShort := Format("{:X}" ,ActiveID) ActiveIDShortLength := StrLen(ActiveIDShort) ActiveTitleLength := StrLen(ActiveTitle) ActiveTitleLength += ActiveIDShortLength ActiveTitleLength += 1 ; +1 the 1 space between title & ID. if ActiveTitleLength > g_MaxLength { ; Since menu item names are limted in length, trim the title ; down to allow just enough room for the Window's Short ID at ; the end of its name: TrimCount := ActiveTitleLength TrimCount -= g_MaxLength ActiveTitle := SubStr(ActiveTitle, 1, -TrimCount) } ; Build unique title: ActiveTitle .= " " ActiveIDShort break } } ; First, ensure that this ID doesn't already exist in the list, which can ; happen if a particular window was externally unhidden (or its app unhid ; it) and now it's about to be re-hidden: AlreadyExists := false for WindowID in g_WindowIDs { if WindowID = ActiveID { AlreadyExists := true break } } ; Add the item to the array and to the menu: if AlreadyExists = false { A_TrayMenu.Add ActiveTitle, RestoreFromTrayMenu g_WindowIDs.Push(ActiveID) g_WindowTitles.Push(ActiveTitle) } } RestoreFromTrayMenu(ThisMenuItem, *) { A_TrayMenu.Delete ThisMenuItem ; Find window based on its unique title stored as the menu item name: for WindowTitle in g_WindowTitles { if WindowTitle = ThisMenuItem ; Match found. { IDToRestore := g_WindowIDs[A_Index] WinShow IDToRestore WinActivate IDToRestore ; Sometimes needed. g_WindowIDs.RemoveAt(A_Index) ; Remove it to free up a slot. g_WindowTitles.RemoveAt(A_Index) break } } } ; This will pop the last minimized window off the stack and unhide it. UnMinimize(*) { ; Make sure there's something to unhide. if g_WindowIDs.Length > 0 { ; Get the id of the last window minimized and unhide it IDToRestore := g_WindowIDs.Pop() WinShow IDToRestore WinActivate IDToRestore ; Get the menu name of the last window minimized and remove it MenuToRemove := g_WindowTitles.Pop() A_TrayMenu.Delete MenuToRemove } } RestoreAllThenExit(*) { RestoreAll() ExitApp ; Do a true exit. } RestoreAll(*) { for WindowID in g_WindowIDs { IDToRestore := WindowID WinShow IDToRestore WinActivate IDToRestore ; Sometimes needed. ; Do it this way vs. DeleteAll so that the sep. line and first ; item are retained: MenuToRemove := g_WindowTitles[A_Index] A_TrayMenu.Delete MenuToRemove } ; Free up all slots: global g_WindowIDs := [] global g_WindowTitles := [] } ï»¿; Changing MsgBox's Button Names ; https://www.autohotkey.com ; This is a working example script that uses a timer to change ; the names of the buttons in a message box. Although the button ; names are changed, the MsgBox's return value still requires that the ; buttons be referred to by their original names. #SingleInstance SetTimer ChangeButtonNames, 50 Result := MsgBox("Choose a button:", "Add or Delete", 4) if Result = "Yes" MsgBox "You chose Add." else MsgBox "You chose Delete." ChangeButtonNames() { if !WinExist("Add or Delete") return ; Keep waiting. SetTimer , 0 WinActivate ControlSetText "&Add", "Button1" ControlSetText "&Delete", "Button2" } ï»¿; Numpad 000 Key ; https://www.autohotkey.com ; This example script makes the special 000 key that appears on certain ; keypads into an equals key. You can change the action by replacing the ; Send "=" line with line(s) of your choice. #MaxThreadsPerHotkey 5 ; Allow multiple threads for this hotkey. $Numpad0:: { #MaxThreadsPerHotkey 1 ; Above: Use the $ to force the hook to be used, which prevents an ; infinite loop since this subroutine itself sends Numpad0, which ; would otherwise result in a recursive call to itself. DelayBetweenKeys := 30 ; Adjust this value if it doesn't work. if A_PriorHotkey = A_ThisHotkey { if A_TimeSincePriorHotkey < DelayBetweenKeys { if Numpad0Count = "" Numpad0Count := 2 ; i.e. This one plus the prior one. else if Numpad0Count = 0 Numpad0Count := 2 else { ; Since we're here, Numpad0Count must be 2 as set by ; prior calls, which means this is the third time the ; the key has been pressed. Thus, the hotkey sequence ; should fire: Numpad0Count := 0 Send "=" ; ******* This is the action for the 000 key } ; In all the above cases, we return without further action: CalledReentrantly := true return } } ; Otherwise, this Numpad0 event is either the first in the series ; or it happened too long after the first one (e.g. perhaps the ; user is holding down the Numpad0 key to auto-repeat it, which ; we want to allow). Therefore, after a short delay -- during ; which another Numpad0 hotkey event may re-entrantly call this ; subroutine -- we'll send the key on through if no reentrant ; calls occurred: Numpad0Count := 0 CalledReentrantly := false ; During this sleep, this subroutine may be reentrantly called ; (i.e. a simultaneous "thread" which runs in parallel to the ; call we're in now): Sleep DelayBetweenKeys if CalledReentrantly = true ; Another "thread" changed the value. { ; Since it was called reentrantly, this key event was the first in ; the sequence so should be suppressed (hidden from the system): CalledReentrantly := false return } ; Otherwise it's not part of the sequence so we send it through normally. ; In other words, the *real* Numpad0 key has been pressed, so we want it ; to have its normal effect: Send "{Numpad0}" } ï»¿; Using Keyboard Numpad as a Mouse (based on the v1 script by deguix) ; https://www.autohotkey.com ; This script makes mousing with your keyboard almost as easy ; as using a real mouse (maybe even easier for some tasks). ; It supports up to five mouse buttons and the turning of the ; mouse wheel. It also features customizable movement speed, ; acceleration, and "axis inversion". /* o----------------------------------------------------------------------o |Using Keyboard Numpad as a Mouse | (----------------------------------------------------------------------) | / A Script file for AutoHotkey | | ----------------------------------------------------------------------| | | | This script is an example of use of AutoHotkey. It uses | | the remapping of numpad keys of a keyboard to transform it | | into a mouse. Some features are the acceleration which | | enables you to increase the mouse movement when holding | | a key for a long time, and the rotation which makes the | | numpad mouse to "turn". I.e. NumpadDown as NumpadUp | | and vice-versa. See the list of keys used below: | | | |-------------------------------------------------------------------| | Keys | Description | |-------------------------------------------------------------------| | ScrollLock (toggle on)| Activates numpad mouse mode. | |----------------------|--------------------------------------------| | Numpad0 | Left mouse button click. | | Numpad5 | Middle mouse button click. | | NumpadDot | Right mouse button click. | | NumpadDiv/NumpadMult | X1/X2 mouse button click. | | NumpadSub/NumpadAdd | Moves up/down the mouse wheel. | | | |----------------------|--------------------------------------------| | NumLock (toggled off) | Activates mouse movement mode. | |----------------------|--------------------------------------------| | NumpadEnd/Down/PgDn/ | Mouse movement. | | /Left/Right/Home/Up/ | | | /PgUp | | | | |----------------------|--------------------------------------------| | NumLock (toggled on) | Activates mouse speed adj. mode. | |----------------------|--------------------------------------------| | Numpad7/Numpad1 | Inc./dec. acceleration per | | | button press. | | Numpad8/Numpad2 | Inc./dec. initial speed per | | | button press. | | Numpad9/Numpad3 | Inc./dec. maximum speed per | | | button press. | | !Numpad7/!Numpad1 | Inc./dec. wheel acceleration per | | | button press*. | | !Numpad8/!Numpad2 | Inc./dec. wheel initial speed per | | | button press*. | | !Numpad9/!Numpad3 | Inc./dec. wheel maximum speed per | | | button press*. | | Numpad4/Numpad6 | Inc./dec. rotation angle to | | | right in degrees. (i.e. 180Â°= | | | = inversed controls). | |-------------------------------------------------------------------| | * = These options are affected by the mouse wheel speed | | adjusted on Control Panel. If you don't have a mouse with | | wheel, the default is 3 +/- lines per option button press. | o----------------------------------------------------------------------o */ ;START OF CONFIG SECTION #SingleInstance A_MaxHotkeysPerInterval := 500 ; Using the keyboard hook to implement the Numpad hotkeys prevents ; them from interfering with the generation of ANSI characters such ; as Â . This is because AutoHotkey generates such characters ; by holding down ALT and sending a series of Numpad keystrokes. ; Hook hotkeys are smart enough to ignore such keystrokes. #UseHook g_MouseSpeed := 1 g_MouseAccelerationSpeed := 1 g_MouseMaxSpeed := 5 ;Mouse wheel speed is also set on Control Panel. As that ;will affect the normal mouse behavior, the real speed of ;these three below are times the normal mouse wheel speed. g_MouseWheelSpeed := 1 g_MouseWheelAccelerationSpeed := 1 g_MouseWheelMaxSpeed := 5 g_MouseRotationAngle := 0 ;END OF CONFIG SECTION ;This is needed or key presses would faulty send their natural ;actions. Like NumpadDiv would send sometimes "/" to the ;screen. InstallKeybdHook g_Temp := 0 g_Temp2 := 0 ;Divide by 45Â° because MouseMove only supports whole numbers, ;and changing the mouse rotation to a number lesser than 45Â° ;could make strange movements. ; ;For example: 22.5Â° when pressing NumpadUp: ; First it would move upwards until the speed ; to the side reaches 1. g_MouseRotationAnglePart := g_MouseRotationAngle / 45 g_MouseCurrentAccelerationSpeed := 0 g_MouseCurrentSpeed := g_MouseSpeed g_MouseCurrentSpeedToDirection := 0 g_MouseCurrentSpeedToSide := 0 g_MouseWheelCurrentAccelerationSpeed := 0 g_MouseWheelCurrentSpeed := g_MouseSpeed g_MouseWheelAccelerationSpeedReal := 0 g_MouseWheelMaxSpeedReal := 0 g_Button := 0 SetKeyDelay -1 SetMouseDelay -1 Hotkey "*Numpad0", ButtonLeftClick Hotkey "*NumpadIns", ButtonLeftClickIns Hotkey "*Numpad5", ButtonMiddleClick Hotkey "*NumpadClear", ButtonMiddleClickClear Hotkey "*NumpadDot", ButtonRightClick Hotkey "*NumpadDel", ButtonRightClickDel Hotkey "*NumpadDiv", ButtonX1Click Hotkey "*NumpadMult", ButtonX2Click Hotkey "*NumpadSub", ButtonWheelAcceleration Hotkey "*NumpadAdd", ButtonWheelAcceleration Hotkey "*NumpadUp", ButtonAcceleration Hotkey "*NumpadDown", ButtonAcceleration Hotkey "*NumpadLeft", ButtonAcceleration Hotkey "*NumpadRight", ButtonAcceleration Hotkey "*NumpadHome", ButtonAcceleration Hotkey "*NumpadEnd", ButtonAcceleration Hotkey "*NumpadPgUp", ButtonAcceleration Hotkey "*NumpadPgDn", ButtonAcceleration Hotkey "Numpad8", ButtonSpeedUp Hotkey "Numpad2", ButtonSpeedDown Hotkey "Numpad7", ButtonAccelerationSpeedUp Hotkey "Numpad1", ButtonAccelerationSpeedDown Hotkey "Numpad9", ButtonMaxSpeedUp Hotkey "Numpad3", ButtonMaxSpeedDown Hotkey "Numpad6", ButtonRotationAngleUp Hotkey "Numpad4", ButtonRotationAngleDown Hotkey "!Numpad8", ButtonWheelSpeedUp Hotkey "!Numpad2", ButtonWheelSpeedDown Hotkey "!Numpad7", ButtonWheelAccelerationSpeedUp Hotkey "!Numpad1", ButtonWheelAccelerationSpeedDown Hotkey "!Numpad9", ButtonWheelMaxSpeedUp Hotkey "!Numpad3", ButtonWheelMaxSpeedDown ToggleKeyActivationSupport ; Initialize based on current ScrollLock state. ;Key activation support ~ScrollLock:: ToggleKeyActivationSupport(*) { ; Wait for it to be released because otherwise the hook state gets reset ; while the key is down, which causes the up-event to get suppressed, ; which in turn prevents toggling of the ScrollLock state/light: KeyWait "ScrollLock" if GetKeyState("ScrollLock", "T") { Hotkey "*Numpad0", "On" Hotkey "*NumpadIns", "On" Hotkey "*Numpad5", "On" Hotkey "*NumpadDot", "On" Hotkey "*NumpadDel", "On" Hotkey "*NumpadDiv", "On" Hotkey "*NumpadMult", "On" Hotkey "*NumpadSub", "On" Hotkey "*NumpadAdd", "On" Hotkey "*NumpadUp", "On" Hotkey "*NumpadDown", "On" Hotkey "*NumpadLeft", "On" Hotkey "*NumpadRight", "On" Hotkey "*NumpadHome", "On" Hotkey "*NumpadEnd", "On" Hotkey "*NumpadPgUp", "On" Hotkey "*NumpadPgDn", "On" Hotkey "Numpad8", "On" Hotkey "Numpad2", "On" Hotkey "Numpad7", "On" Hotkey "Numpad1", "On" Hotkey "Numpad9", "On" Hotkey "Numpad3", "On" Hotkey "Numpad6", "On" Hotkey "Numpad4", "On" Hotkey "!Numpad8", "On" Hotkey "!Numpad2", "On" Hotkey "!Numpad7", "On" Hotkey "!Numpad1", "On" Hotkey "!Numpad9", "On" Hotkey "!Numpad3", "On" } else { Hotkey "*Numpad0", "Off" Hotkey "*NumpadIns", "Off" Hotkey "*Numpad5", "Off" Hotkey "*NumpadDot", "Off" Hotkey "*NumpadDel", "Off" Hotkey "*NumpadDiv", "Off" Hotkey "*NumpadMult", "Off" Hotkey "*NumpadSub", "Off" Hotkey "*NumpadAdd", "Off" Hotkey "*NumpadUp", "Off" Hotkey "*NumpadDown", "Off" Hotkey "*NumpadLeft", "Off" Hotkey
```

```
"*NumpadRight", "Off" Hotkey "*NumpadHome", "Off" Hotkey "*NumpadEnd", "Off" Hotkey "*NumpadPgUp", "Off" Hotkey "*NumpadPgDn", "Off" Hotkey "Numpad8", "Off" Hotkey "Numpad2", "Off" Hotkey "Numpad7", "Off" Hotkey "Numpad1", "Off" Hotkey "Numpad9", "Off" Hotkey "Numpad3", "Off" Hotkey "Numpad6", "Off" Hotkey "Numpad4", "Off" Hotkey "!Numpad8", "Off" Hotkey "!Numpad2", "Off" Hotkey "!Numpad7", "Off" Hotkey "!Numpad1", "Off" Hotkey "!Numpad9", "Off" Hotkey "!Numpad3", "Off" } } ;Mouse click support ButtonLeftClick(*) { if GetKeyState("LButton") return Button2 := "Numpad0" ButtonClick := "Left" ButtonClickStart Button2, ButtonClick } ButtonLeftClickIns(*) { if GetKeyState("LButton") return Button2 := "NumpadIns" ButtonClick := "Left" ButtonClickStart Button2, ButtonClick } ButtonMiddleClick(*) { if GetKeyState("MButton") return Button2 := "Numpad5" ButtonClick := "Middle" ButtonClickStart Button2, ButtonClick } ButtonMiddleClickClear(*) { if GetKeyState("MButton") return Button2 := "NumpadClear" ButtonClick := "Middle" ButtonClickStart Button2, ButtonClick } ButtonRightClick(*) { if GetKeyState("RButton") return Button2 := "NumpadDot" ButtonClick := "Right" ButtonClickStart Button2, ButtonClick } ButtonRightClickDel(*) { if GetKeyState("RButton") return Button2 := "NumpadDel" ButtonClick := "Right" ButtonClickStart Button2, ButtonClick } ButtonX1Click(*) { if GetKeyState("XButton1") return Button2 := "NumpadDiv" ButtonClick := "X1" ButtonClickStart Button2, ButtonClick } ButtonX2Click(*) { if GetKeyState("XButton2") return Button2 := "NumpadMult" ButtonClick := "X2" ButtonClickStart Button2, ButtonClick } ButtonClickStart (Button2, ButtonClick) { MouseClick ButtonClick,,, 1, 0, "D" SetTimer ButtonClickEnd.Bind(Button2, ButtonClick), 10 } ButtonClickEnd(Button2, ButtonClick) { if GetKeyState(Button2, "P") return SetTimer , 0 MouseClick ButtonClick,,, 1, 0, "U" } ;Mouse movement support ButtonSpeedUp(*) { global g_MouseSpeed++ ToolTip "Mouse speed: " g_MouseSpeed " pixels" SetTimer ToolTip, -1000 } ButtonSpeedDown(*) { global if g_MouseSpeed > 1 g_MouseSpeed-- if g_MouseSpeed = 1 ToolTip "Mouse speed: " g_MouseSpeed " pixel" else ToolTip "Mouse speed: " g_MouseSpeed " pixels" SetTimer ToolTip, -1000 } ButtonAccelerationSpeedUp(*) { global g_MouseAccelerationSpeed++ ToolTip "Mouse acceleration speed: " g_MouseAccelerationSpeed " pixels" SetTimer ToolTip, -1000 } ButtonAccelerationSpeedDown(*) { global if g_MouseAccelerationSpeed > 1 g_MouseAccelerationSpeed-- if g_MouseAccelerationSpeed = 1 ToolTip "Mouse acceleration speed: " g_MouseAccelerationSpeed " pixel" else ToolTip "Mouse acceleration speed: " g_MouseAccelerationSpeed " pixels" SetTimer ToolTip, -1000 } ButtonMaxSpeedUp(*) { global g_MouseMaxSpeed++ ToolTip "Mouse maximum speed: " g_MouseMaxSpeed " pixels" SetTimer ToolTip, -1000 } ButtonMaxSpeedDown(*) { global if g_MouseMaxSpeed > 1 g_MouseMaxSpeed-- if g_MouseMaxSpeed = 1 ToolTip "Mouse maximum speed: " g_MouseMaxSpeed " pixel" else ToolTip "Mouse maximum speed: " g_MouseMaxSpeed " pixels" SetTimer ToolTip, -1000 } ButtonRotationAngleUp(*) { global g_MouseRotationAnglePart++ if g_MouseRotationAnglePart >= 8 g_MouseRotationAnglePart := 0 g_MouseRotationAngle := g_MouseRotationAnglePart g_MouseRotationAngle *= 45 ToolTip "Mouse rotation angle: " g_MouseRotationAngle "Â°" SetTimer ToolTip, -1000 } ButtonRotationAngleDown(*) { global g_MouseRotationAnglePart-- if g_MouseRotationAnglePart < 0 g_MouseRotationAnglePart := 7 g_MouseRotationAngle := g_MouseRotationAnglePart g_MouseRotationAngle *= 45 ToolTip "Mouse rotation angle: " g_MouseRotationAngle "Â°" SetTimer ToolTip, -1000 } ButtonAcceleration(ThisHotkey) { global if g_Button != 0 { if !InStr(ThisHotkey, g_Button) { g_MouseCurrentAccelerationSpeed := 0 g_MouseCurrentSpeed := g_MouseSpeed } } g_Button := StrReplace(ThisHotkey, "*") ButtonAccelerationStart } ButtonAccelerationStart() { global if g_MouseAccelerationSpeed >= 1 { if g_MouseMaxSpeed > g_MouseCurrentSpeed { g_Temp := 0.001 g_Temp *= g_MouseAccelerationSpeed g_MouseCurrentAccelerationSpeed += g_Temp g_MouseCurrentSpeed += g_MouseCurrentAccelerationSpeed } } ;g_MouseRotationAngle convertion to speed of button direction g_MouseCurrentSpeedToDirection := g_MouseRotationAngle g_MouseCurrentSpeedToDirection /= 90.0 g_Temp := g_MouseCurrentSpeedToDirection if g_Temp >= 0 { if g_Temp < 1 { g_MouseCurrentSpeedToDirection := 1 g_MouseCurrentSpeedToDirection -= g_Temp EndMouseCurrentSpeedToDirectionCalculation return } } if g_Temp >= 1 { if g_Temp < 2 { g_MouseCurrentSpeedToDirection := 0 g_Temp -= 1 g_MouseCurrentSpeedToDirection -= g_Temp EndMouseCurrentSpeedToDirectionCalculation return } } if g_Temp >= 2 { if g_Temp < 3 { g_MouseCurrentSpeedToDirection := -1 g_Temp -= 2 g_MouseCurrentSpeedToDirection += g_Temp EndMouseCurrentSpeedToDirectionCalculation return } } if g_Temp >= 3 { if g_Temp < 4 { g_MouseCurrentSpeedToDirection := 0 g_Temp -= 3 g_MouseCurrentSpeedToDirection += g_Temp EndMouseCurrentSpeedToDirectionCalculation return } } EndMouseCurrentSpeedToDirectionCalculation } EndMouseCurrentSpeedToDirectionCalculation() { global ;g_MouseRotationAngle convertion to speed of 90 degrees to right g_MouseCurrentSpeedToSide := g_MouseRotationAngle g_MouseCurrentSpeedToSide /= 90.0 g_Temp := Mod(g_MouseCurrentSpeedToSide, 4) if g_Temp >= 0 { if g_Temp < 1 { g_MouseCurrentSpeedToSide := 0 g_MouseCurrentSpeedToSide += g_Temp EndMouseCurrentSpeedToSideCalculation return } } if g_Temp >= 1 { if g_Temp < 2 { g_MouseCurrentSpeedToSide := 1 g_Temp -= 1 g_MouseCurrentSpeedToSide -= g_Temp EndMouseCurrentSpeedToSideCalculation return } } if g_Temp >= 2 { if g_Temp < 3 { g_MouseCurrentSpeedToSide := 0 g_Temp -= 2 g_MouseCurrentSpeedToSide -= g_Temp EndMouseCurrentSpeedToSideCalculation return } } if g_Temp >= 3 { if g_Temp < 4 { g_MouseCurrentSpeedToSide := -1 g_Temp -= 3 g_MouseCurrentSpeedToSide += g_Temp EndMouseCurrentSpeedToSideCalculation return } } EndMouseCurrentSpeedToSideCalculation } EndMouseCurrentSpeedToSideCalculation() { global g_MouseCurrentSpeedToDirection *= g_MouseCurrentSpeed g_MouseCurrentSpeedToSide *= g_MouseCurrentSpeed g_Temp := Mod(g_MouseRotationAnglePart, 2) if g_Button = "NumpadUp" { if g_Temp = 1 { g_MouseCurrentSpeedToSide *= 2 g_MouseCurrentSpeedToDirection *= 2 } g_MouseCurrentSpeedToDirection *= -1 MouseMove g_MouseCurrentSpeedToSide, g_MouseCurrentSpeedToDirection, 0, "R" } else if g_Button = "NumpadDown" { if g_Temp = 1 { g_MouseCurrentSpeedToSide *= 2 g_MouseCurrentSpeedToDirection *= 2 } g_MouseCurrentSpeedToSide *= -1 MouseMove g_MouseCurrentSpeedToSide, g_MouseCurrentSpeedToDirection, 0, "R" } else if g_Button = "NumpadLeft" { if g_Temp = 1 { g_MouseCurrentSpeedToSide *= 2 g_MouseCurrentSpeedToDirection *= 2 } g_MouseCurrentSpeedToSide *= -1 g_MouseCurrentSpeedToDirection *= -1 MouseMove g_MouseCurrentSpeedToDirection, g_MouseCurrentSpeedToSide, 0, "R" } else if g_Button = "NumpadRight" { if g_Temp = 1 { g_MouseCurrentSpeedToSide *= 2 g_MouseCurrentSpeedToDirection *= 2 } MouseMove g_MouseCurrentSpeedToDirection, g_MouseCurrentSpeedToSide, 0, "R" } else if g_Button = "NumpadHome" { g_Temp := g_MouseCurrentSpeedToDirection g_Temp -= g_MouseCurrentSpeedToSide g_Temp *= -1 g_Temp2 := g_MouseCurrentSpeedToDirection g_Temp2 += g_MouseCurrentSpeedToSide g_Temp2 *= -1 MouseMove g_Temp, g_Temp2, 0, "R" } else if g_Button = "NumpadPgUp" { g_Temp := g_MouseCurrentSpeedToDirection g_Temp += g_MouseCurrentSpeedToSide g_Temp2 := g_MouseCurrentSpeedToDirection g_Temp2 -= g_MouseCurrentSpeedToSide g_Temp2 *= -1 MouseMove g_Temp, g_Temp2, 0, "R" } else if g_Button = "NumpadEnd" { g_Temp := g_MouseCurrentSpeedToDirection g_Temp += g_MouseCurrentSpeedToSide g_Temp *= -1 g_Temp2 := g_MouseCurrentSpeedToDirection g_Temp2 -= g_MouseCurrentSpeedToSide MouseMove g_Temp, g_Temp2, 0, "R" } else if g_Button = "NumpadPgDn" { g_Temp := g_MouseCurrentSpeedToDirection g_Temp -= g_MouseCurrentSpeedToSide g_Temp2 *= -1 g_Temp2 := g_MouseCurrentSpeedToDirection g_Temp2 += g_MouseCurrentSpeedToSide MouseMove g_Temp, g_Temp2, 0, "R" } SetTimer ButtonAccelerationEnd, 10 } ButtonAccelerationEnd() { global if GetKeyState(g_Button, "P") { ButtonAccelerationStart return } SetTimer , 0 g_MouseCurrentAccelerationSpeed := 0 g_MouseCurrentSpeed := g_MouseSpeed g_Button := 0 } ;Mouse wheel movement support ButtonWheelSpeedUp(*) { global g_MouseWheelSpeed++ local MouseWheelSpeedMultiplier := RegRead("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 g_MouseWheelSpeedReal := g_MouseWheelSpeed g_MouseWheelSpeedReal *= MouseWheelSpeedMultiplier ToolTip "Mouse wheel speed: " g_MouseWheelSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelSpeedDown(*) { global local MouseWheelSpeedMultiplier := RegRead ("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 if g_MouseWheelSpeedReal > MouseWheelSpeedMultiplier { g_MouseWheelSpeed-- g_MouseWheelSpeedReal := g_MouseWheelSpeed g_MouseWheelSpeedReal *= MouseWheelSpeedMultiplier } if g_MouseWheelSpeedReal = 1 ToolTip "Mouse wheel speed: " g_MouseWheelSpeedReal " line" else ToolTip "Mouse wheel speed: " g_MouseWheelSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelAccelerationSpeedUp(*) { global g_MouseWheelAccelerationSpeed++ local MouseWheelSpeedMultiplier := RegRead ("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 g_MouseWheelAccelerationSpeedReal := g_MouseWheelAccelerationSpeed g_MouseWheelAccelerationSpeedReal *= MouseWheelSpeedMultiplier ToolTip "Mouse wheel acceleration speed: " g_MouseWheelAccelerationSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelAccelerationSpeedDown(*) { global local MouseWheelSpeedMultiplier := RegRead("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 if g_MouseWheelAccelerationSpeed > 1 { g_MouseWheelAccelerationSpeed-- g_MouseWheelAccelerationSpeedReal := g_MouseWheelAccelerationSpeed g_MouseWheelAccelerationSpeedReal *= MouseWheelSpeedMultiplier } if g_MouseWheelAccelerationSpeedReal = 1 ToolTip "Mouse wheel acceleration speed: " g_MouseWheelAccelerationSpeedReal " line" else ToolTip "Mouse wheel acceleration speed: " g_MouseWheelAccelerationSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelMaxSpeedUp(*) { global g_MouseWheelMaxSpeed++ local MouseWheelSpeedMultiplier := RegRead("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 g_MouseWheelMaxSpeedReal := g_MouseWheelMaxSpeed g_MouseWheelMaxSpeedReal *= MouseWheelSpeedMultiplier ToolTip "Mouse wheel maximum speed: " g_MouseWheelMaxSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelMaxSpeedDown(*) { global local MouseWheelSpeedMultiplier := RegRead("HKCU\Control Panel\Desktop", "WheelScrollLines") if MouseWheelSpeedMultiplier <= 0 MouseWheelSpeedMultiplier := 1 if g_MouseWheelMaxSpeed > 1 { g_MouseWheelMaxSpeed-- g_MouseWheelMaxSpeedReal := g_MouseWheelMaxSpeed g_MouseWheelMaxSpeedReal *= MouseWheelSpeedMultiplier } if g_MouseWheelMaxSpeedReal = 1 ToolTip "Mouse wheel maximum speed: " g_MouseWheelMaxSpeedReal " line" else ToolTip "Mouse wheel maximum speed: " g_MouseWheelMaxSpeedReal " lines" SetTimer ToolTip, -1000 } ButtonWheelAcceleration(ThisHotkey) { global if g_Button != 0 { if g_Button != ThisHotkey { g_MouseWheelCurrentAccelerationSpeed := 0 g_MouseWheelCurrentSpeed := g_MouseWheelSpeed } } g_Button := StrReplace(ThisHotkey, "*") ButtonWheelAccelerationStart } ButtonWheelAccelerationStart() { global if g_MouseWheelAccelerationSpeed >= 1 { if g_MouseWheelMaxSpeed > g_MouseWheelCurrentSpeed { g_Temp := 0.001 g_Temp *= g_MouseWheelAccelerationSpeed g_MouseWheelCurrentAccelerationSpeed += g_Temp g_MouseWheelCurrentSpeed += g_MouseWheelCurrentAccelerationSpeed } } if g_Button = "NumpadSub" MouseClick "WheelUp",,, g_MouseWheelCurrentSpeed, 0, "D" else if g_Button = "NumpadAdd" MouseClick "WheelDown",,, g_MouseWheelCurrentSpeed, 0, "D" SetTimer ButtonWheelAccelerationEnd, 100 } ButtonWheelAccelerationEnd() { global if GetKeyState(g_Button, "P")
```

{ ButtonWheelAccelerationStart return } g_MouseWheelCurrentAccelerationSpeed := 0 g_MouseWheelCurrentSpeed := g_MouseWheelSpeed g_Button := 0 } ï»¿/ Seek (based on the v1 script by Phi) http://www.autohotkey.com Navigating the Start Menu can be a hassle, especially if you have installed many programs over time. 'Seek' lets you specify a case-insensitive key word/phrase that it will use to filter only the matching programs and directories from the Start Menu, so that you can easily open your target program from a handful of matched entries. This eliminates the drudgery of searching and traversing the Start Menu. */ /* Options: -cache Use the cached directory-listing if available (this is the default mode when no option is specified) -scan Force a directory scan to retrieve the latest directory listing -scex Scan & exit (this is useful for scheduling the potentially time-consuming directory-scanning as a background job) -help Show this help Important notes: Check AutoHotkey's Tutorial how to run this script, or to compile it if necessary. The only file 'Seek' creates is placed in your TMP directory: a. _Seek.ini (cache file for last query string and directory listing) When you run 'Seek' for the first time, it'll scan your Start Menu, and save the directory listing into a cache file. The following directories are included in the scanning: - A_StartMenu - A_StartMenuCommon By default, subsequent runs will read from the cache file so as to reduce the loading time. For more info on options, run 'Seek.exe -help'. If you think your Start Menu doesn't contain too many programs, you can choose not to use the cache and instruct 'Seek' to always do a directory scan (via option -scan). That way, you will always get the latest listing. When you run 'Seek', a window will appear, waiting for you to enter a key word/phrase. After you have entered a query string, a list of matching records will be displayed. Next, you need to highlight an entry and press ENTER or click on the 'Open' button to run the selected program or open the selected directory. */ /* Specify which program to use when opening a directory. If the program cannot be found or is not specified (i.e. variable is unassigned or assigned a null value), the default Explorer will be used. */ g_dirExplorer := "E:\utl\xplorer2_lite\xplorer2.exe" /* User's customised list of additional directories to be included in the scanning. The full path must not be enclosed by quotes or double-quotes. If this file is missing, only the default directories will be scanned. */ g_SeekMyDir := A_ScriptDir "\Seek.dir" /* Specify the filename and directory location to save the cached directory/program listing and the cached key word/phrase of the last search. There is no need to change this unless you want to. */ g_saveFile := A_Temp "_Seek.ini" /* Track search string (True/False) If true, the last-used query string will be re-used as the default query string the next time you run Seek. If false, the last-used query string will not be tracked and there will not be a default query string value the next time you run Seek. */ g_TrackKeyPhrase := True /* Specify what should be included in scan. F: Files are included. D: Directories are included. */ g_ScanMode := "FD" ; Init: ; #NoTrayIcon ; Define the script title: g_ScriptTitle := "Seek - Search the Start Menu" ; Display the help instructions: if (A_Args.Length && A_Args[1] ~= "^(--help|-help|/h|-h|\?|-\?)$") { MsgBox(" (Navigating the Start Menu can be a hassle, especially if you have installed many programs over time. 'Seek' lets you specify a case-insensitive key word/phrase that it will use to filter only the matching programs and directories from the Start Menu, so that you can easily open your target program from a handful of matched entries. This eliminates the drudgery of searching and traversing the Start Menu. Options: -cache Use the cached directory-listing if available (this is the default mode when no option is specified) -scan Force a directory scan to retrieve the latest directory listing -scex Scan & exit (this is useful for scheduling the potentially time-consuming directory-scanning as a background job) -help Show this help)", g_ScriptTitle) ExitApp } ; Check that the mandatory environment variables exist and are valid: if !DirExist(A_Temp) ; Path does not exist. { MsgBox ("This mandatory environment variable is either not defined or invalid: TMP = " A_Temp " Please fix it before running Seek."), g_ScriptTitle ExitApp } ; Scan the Start Menu without Gui: if (A_Args.Length && A_Args[1] = "-scex") { SaveFileList() return } ; Create the GUI window: G := Gui(, g_ScriptTitle) ; Add the text box for user to enter the query string: E_Search := G.Add("Edit", "W600") E_Search.OnEvent("Change", FindMatches) if g_TrackKeyPhrase try E_Search.Value := IniRead(g_saveFile, "LastSession", "SearchText") ; Add my fav tagline: G.Add("Text", "X625 Y10", "What do you seek, my friend?") ; Add the status bar for providing feedback to user: T_Info := G.Add("Text", "X10 Y31 R1 W764") ; Add the selection listbox for displaying search results: LB := G.Add("ListBox", "X10 Y53 R28 W764 HScroll Disabled") LB.OnEvent("DoubleClick", OpenTarget) ; Add these buttons, but disable them for now: B1 := G.Add("Button", "Default X10 Y446 Disabled", "Open") B1.OnEvent("Click", OpenTarget) B2 := G.Add("Button", "X59 Y446 Disabled", "Open Directory") B2.OnEvent("Click", OpenFolder) B3 := G.Add("Button", "X340 Y446", "Scan Start-Menu") B3.OnEvent ("Click", ScanStartMenu) ; Add the Exit button: G.Add("Button", "X743 Y446", "Exit").OnEvent("Click", (*) => Gui_Close(G)) ; Add window events: G.OnEvent ("Close", Gui_Close) G.OnEvent("Escape", Gui_Close) ; Pop-up the query window: G.Show("Center") ; Force re-scanning if -scan is enabled or listing cache file does not exist: if (A_Args.Length && A_Args[1] = "-scan" || !FileExist(g_saveFile)) ScanStartMenu() ; Retrieve an set the matching list: FindMatches() ; Retrieve the last selection from cache file and select the item: if g_TrackKeyPhrase try if (LastSelection := IniRead(g_saveFile, "LastSession", "Selection")) LB.Choose(LastSelection) ; Function definitions --- ; Scan the start-menu and store the directory/program listings in a cache file: ScanStartMenu(*) { ; Inform user that scanning has started: T_Info.Value := "Scanning directory listing..." ; Disable listbox while scanning is in progress: LB.Enabled := false B1.Enabled := false B2.Enabled := false B3.Enabled := false ; Retrieve and save the start menu files: SaveFileList() ; Inform user that scanning has completed: T_Info.Value := "Scan completed." ; Enable listbox: LB.Enabled := true B1.Enabled := true B2.Enabled := true B3.Enabled := true ; Filter for search string with the new listing: FindMatches() } ; Retrieve and save the start menu files: SaveFileList() { ; Define the directory paths to retrieve: LocationArray := [A_StartMenu, A_StartMenuCommon] ; Include additional user-defined paths for scanning: if FileExist(g_SeekMyDir) Loop Read, g_SeekMyDir { if !DirExist(A_LoopReadLine) MsgBox ("Processing your customised directory list... '" A_LoopReadLine "' does not exist and will be excluded from the scanning. Please update [" g_SeekMyDir "]."), g_ScriptTitle, 8192 else LocationArray.Push (A_LoopReadLine) } ; Scan directory listing by recursing each directory to retrieve the contents. ; Hidden files are excluded: IniDelete(g_saveFile, "LocationList") For i, Location in LocationArray { ; Save space by using relative paths: IniWrite(Location, g_saveFile, "LocationList", "L" i) A_WorkingDir := Location FileList := "" Loop Files, "*", g_ScanMode "R" if !InStr(FileGetAttrib(A_LoopFilePath), "H") ; Exclude hidden file. FileList .= "%L" i "%\" A_LoopFilePath "`n" } IniDelete (g_saveFile, "FileList") IniWrite(FileList, g_saveFile, "FileList") } ; Search and display all matching records in the listbox: FindMatches(*) { FileArray := [] SearchText := E_Search.Value ; Filter matching records based on user query string: if SearchText { L := Map() while (Location := IniRead(g_saveFile, "LocationList", "L" A_Index, "")) L[A_Index] := Location Loop Parse, IniRead(g_saveFile, "FileList"), "`n" { Line := A_LoopField if RegExMatch(Line, "%L(\d+)%", &m) ; Replace %L1% etc. with location paths. Line := StrReplace(Line, "%L" m[1] "%", L[Integer(m[1])]) if (SearchText != E_Search.Value) { ; User has changed the search string. ; There is no point to continue searching using the old string, so abort. return } else { ; Append matching records into the list: SplitPath(Line, &Name) MatchFound := true Loop Parse, SearchText, "`s" { if !InStr(Name, A_LoopField) { MatchFound := false break } } if MatchFound FileArray.Push(Line) } } } ; Refresh list with search results: LB.Delete(), LB.Add(FileArray) if !FileArray.Length { ; No matching record is found. Disable listbox: LB.Enabled := false B1.Enabled := false B2.Enabled := false } else { ; Matching records are found. Enable listbox: LB.Enabled := true B1.Enabled := true B2.Enabled := true ; Select the first record if no other record has been selected: if (LB.Text = "") LB.Choose(1) } } ; User clicked on 'Open' button or pressed ENTER: OpenTarget(*) { ; Selected record does not exist (file or directory not found): if !FileExist(LB.Text) { MsgBox (LB.Text " does not exist. This means that the directory cache is outdated. You may click on the 'Scan Start-Menu' button below to update the directory cache with your latest directory listing now."), g_ScriptTitle, 8192 return } ; Check whether the selected record is a file or directory: fileAttrib := FileGetAttrib(LB.Text) if InStr(fileAttrib, "D") ; is directory OpenFolder() else if fileAttrib ; is file Run(LB.Text) else { MsgBox (LB.Text " is neither a DIRECTORY or a FILE. This shouldn't happen. Seek cannot proceed. Quitting...") } WinClose } ; User clicked on 'Open Directory' button: OpenFolder(*) { Path := LB.Text ; If user selected a file-record instead of a directory-record, extract the ; directory path (I'm using DriveGetStatus instead of FileGetAttrib to allow the ; scenario whereby LB.Text is invalid but the directory path of LB.Text is valid): if (DriveGetStatus(Path) != "Ready") ; not a directory { SplitPath(Path,, &Dir) Path := Dir } ; Check whether directory exists: if !DirExist(Path) { MsgBox (Path " does not exist. This means that the directory cache is outdated. You may click on the 'Scan Start-Menu' button below to update the directory cache with your latest directory listing now."), g_ScriptTitle, 8192 return } ; Open the directory: if FileExist(g_dirExplorer) Run(Format('"{1}" "{2}"', g_dirExplorer, Path)) ; Open with custom file explorer. else Run(Path) ; Open with default windows file explorer. } Gui_Close(*) { ; Save the key word/phrase for next run: if g_TrackKeyPhrase { IniWrite(E_Search.Value, g_saveFile, "LastSession", "SearchText") IniWrite(LB.Text, g_saveFile, "LastSession", "Selection") } ExitApp } ï»¿; ToolTip Mouse Menu (based on the v1 script by Rajat) ; https://www.autohotkey.com ; This script displays a popup menu in response to briefly holding down ; the middle mouse button. Select a menu item by left-clicking it. ; Cancel the menu by left-clicking outside of it. A recent improvement ; is that the contents of the menu can change depending on which type of ; window is active (Notepad and Word are used as examples here). ; You can set any title here for the menu: g_MenuTitle := "-=-=-=-=-=-=-=-" ; This is how long the mouse button must be held to cause the menu to appear: g_UMDelay := 20 #SingleInstance ; _____ ; _____Menu Definitions_____ ; Create / Edit Menu Items here. ; You can't use spaces in keys/values/section names. ; Don't worry about the order, the menu will be sorted. g_MenuItems := "Notepad/Calculator/Section 3/Section 4/Section 5" ; _____ ; _____Dynamic menuitems here_____ ; Syntax: ; "MenuItem|Window title" g_Dyn := ["MS Word|- Microsoft Word", "Notepad II|- Notepad",] ; _____ Exit ; _____ ; _____Menu Sections_____ ; Create / Edit Menu Sections here. Notepad() { Run "Notepad.exe" } Calculator() { Run "Calc" } Section3() { MsgBox "You selected 3" } Section4() { MsgBox "You selected 4" } Section5() { MsgBox "You selected 5" } MSWord() { MsgBox "this is a dynamic entry (word)" } NotepadII() { MsgBox "this is a dynamic entry (notepad)" } ; _____ ; _____Hotkey Section_____ ~MButton:: { HowLong := 0 Loop { HowLong++ Sleep 10 if !GetKeyState("MButton", "P") Break } if HowLong < g_UMDelay return ; Prepares dynamic menu: DynMenu := "" For i, item in g_Dyn { mp := StrSplit (item, "|") if WinActive(mp[2]) DynMenu .= "/" mp[1] } ; Joins sorted main menu and dynamic menu, and ; clears earlier entries and creates new entries: MenuItem := StrSplit(Sort(g_MenuItems, "D/") DynMenu, "/") ; Creates the menu: ToolTipMenu := g_MenuTitle For i, item in MenuItem ToolTipMenu .= "`n" item MouseGetPos &mX, &mY Hotkey "~LButton", MenuClick Hotkey "~LButton", "On" ToolTip ToolTipMenu, mX, mY WinActivate g_MenuTitle WinGetPos ,,, &tH, g_MenuTitle MenuClick(*) { Hotkey "~LButton", "Off" if !WinActive(g_MenuTitle) { ToolTip return } MouseGetPos &mX, &mY ToolTip mY /= tH / (MenuItem.Length + 1) ; Space taken by each line. if mY < 1 return TargetSection := MenuItem[Integer(mY)] %StrReplace(TargetSection, "`s")%() } } ï»¿/* Volume On-Screen-Display (based on the v1 script by Rajat) https://www.autohotkey.com This script assigns hotkeys of your choice to raise and lower the master wave volume. */ ; --- User Settings --- ; The*

*percentage by which to raise or lower the volume each time: g_Step := 4 ; How long to display the volume level bar graphs: g_DisplayTime := 2000 ; Master Volume Bar color (see the help file to use more precise shades): g_CBM := "Red" ; Background color: g_CW := "Silver" ; Bar's screen position. Use "center" to center the bar in that dimension: g_PosX := "center" g_PosY := "center" g_Width := 150 ; width of bar g_Thick := 12 ; thickness of bar ; If your keyboard has multimedia buttons for Volume, you can ; try changing the below hotkeys to use them by specifying ; Volume_Up and Volume_Down: g_MasterUp := "#Up" ; Win+UpArrow g_MasterDown := "#Down" ; --- Auto Execute Section --- ; DON'T CHANGE ANYTHING HERE (unless you know what you're doing). #SingleInstance ; Create the Progress window: G := Gui("+ToolWindow -Caption -Border +Disabled") G.MarginX := 0, G.MarginY := 0 opt := "w" g_Width " h" g_Thick " background" g_CW Master := G.Add ("Progress", opt " c" g_CBM) ; Register hotkeys: Hotkey g_MasterUp, (*) => ChangeVolume("+") Hotkey g_MasterDown, (*) => ChangeVolume("-") ; --- Function Definitions --- ChangeVolume(Prefix) { SoundSetVolume(Prefix g_Step) Master.Value := Round(SoundGetVolume()) G.Show("x" g_PosX " y" g_PosY) SetTimer HideWindow, -g_DisplayTime } HideWindow() { G.Hide() } ï»¿; Window Shading (based on the v1 script by Rajat) ; https://www.autohotkey.com ; This script reduces a window to its title bar and then back to its ; original size by pressing a single hotkey. Any number of windows ; can be reduced in this fashion (the script remembers each). If the ; script exits for any reason, all "rolled up" windows will be ; automatically restored to their original heights. ; Set the height of a rolled up window here. The operating system ; probably won't allow the title bar to be hidden regardless of ; how low this number is: g_MinHeight := 25 ; This line will unroll any rolled up windows if the script exits ; for any reason: OnExit ExitSub IDs := Array() Windows := Map() #z:: ; Change this line to pick a different hotkey. ; Below this point, no changes should be made unless you want to ; alter the script's basic functionality. { ; Uncomment this next line if this subroutine is to be converted ; into a custom menu item rather than a hotkey. The delay allows ; the active window that was deactivated by the displayed menu to ; become active again: ;Sleep 200 ActiveID := WinGetID ("A") for ID in IDs { if ID = ActiveID { ; Match found, so this window should be restored (unrolled): Height := Windows[ActiveID] WinMove ,,, Height, ActiveID IDs.RemoveAt(A_Index) return } } WinGetPos ,,, &Height, "A" Windows.Set(ActiveID, Height) WinMove ,,, g_MinHeight, ActiveID IDs.Push(ActiveID) } ExitSub(*) { for ID in IDs { Height := Windows[ID] WinMove ,,, Height, ID } ExitApp ; Must do this for the OnExit subroutine to actually Exit the script. } ï»¿; WinLIRC Client ; https://www.autohotkey.com ; This script receives notifications from WinLIRC whenever you press ; a button on your remote control. It can be used to automate Winamp, ; Windows Media Player, etc. It's easy to configure. For example, if ; WinLIRC recognizes a button named "VolUp" on your remote control, ; create a label named VolUp and beneath it use the function ; to increase the soundcard's volume by 5%. ; Here are the steps to use this script: ; 1) Configure WinLIRC to recognize your remote control and its buttons. ; WinLIRC is at http://winlirc.sourceforge.net ; 2) Edit the WinLIRC path, address, and port in the CONFIG section below. ; 3) Launch this script. It will start the WinLIRC server if needed. ; 4) Press some buttons on your remote control. A small window will ; appear showing the name of each button as you press it. ; 5) Configure your buttons to send keystrokes and mouse clicks to ; windows such as Winamp, Media Player, etc. See the examples below. ; ---------- ---------- ; CONFIGURATION SECTION: Set your preferences here. ; ---------- ; Some remote controls repeat the signal rapidly while you're holding down ; a button. This makes it difficult to get the remote to send only a single ; signal. The following setting solves this by ignoring repeated signals ; until the specified time has passed. 200 is often a good setting. Set it ; to 0 to disable this feature. g_DelayBetweenButtonRepeats := 200 ; Specify the path to WinLIRC, such as C:\WinLIRC\winlirc.exe WinLIRC_Path := A_ProgramFiles "\WinLIRC\winlirc.exe" ; Specify WinLIRC's address and port. The most common are 127.0.0.1 (localhost) and 8765. WinLIRC_Address := "127.0.0.1" WinLIRC_Port := "8765" ; Do not change the following line. Skip it and continue below. WinLIRC_Init(WinLIRC_Path, WinLIRC_Address, WinLIRC_Port) ; ---------- ; ASSIGN ACTIONS TO THE BUTTONS ON YOUR REMOTE ; ---------- ; Configure your remote control's buttons below. Use WinLIRC's names ; for the buttons, which can be seen in your WinLIRC config file ; (.cf file) -- or you can press any button on your remote and the ; script will briefly display the button's name in a small window. ; ; Below are some examples. Feel free to revise or delete them to suit ; your preferences. class Actions { VolUp() { SoundSetVolume "+5" ; Increase master volume by 5%. } VolDown() { SoundSetVolume "-5" ; Reduce master volume by 5%. } ChUp() { if WinGetClass("A") ~= "^(Winamp v1\.x|Winamp PE)$" ; Winamp is active. Send "{right}" ; Send a right-arrow keystroke. else ; Some other type of window is active. Send "{WheelUp}" ; Rotate the mouse wheel up by one notch. } ChDown() { if WinGetClass("A") ~= "^(Winamp v1\.x|Winamp PE)$" ; Winamp is active. Send "{left}" ; Send a left-arrow keystroke. else ; Some other type of window is active. Send "{WheelDown}" ; Rotate the mouse wheel down by one notch. } Menu() { if WinExist("Untitled - Notepad") { WinActivate } else { Run "Notepad" WinWait "Untitled - Notepad" WinActivate } Send "Here are some keystrokes sent to Notepad.{Enter}" } } ; The examples above give a feel for how to accomplish common tasks. ; To learn the basics of AutoHotkey, check out the Quick-start Tutorial ; at https://www.autohotkey.com/docs/Tutorial.htm ; ---------- ; END OF CONFIGURATION SECTION ; ---------- ; Do not make changes below this point unless you want to change the core ; functionality of the script. WinLIRC_Init(Path, IPAddress, Port) { OnExit ExitSub ; For connection cleanup purposes. ; Launch WinLIRC if it isn't already running: if not ProcessExist("winlirc.exe") ; No PID for WinLIRC was found. { if !FileExist(Path) { MsgBox "The file '" Path "' does not exist. Please edit this script to specify its location." ExitApp } Run Path Sleep 200 ; Give WinLIRC a little time to initialize (probably never needed, just for peace of mind). } ; Connect to WinLIRC (or any type of server for that matter): socket := ConnectToAddress (IPAddress, Port) if socket = -1 ; Connection failed (it already displayed the reason). ExitApp ; When the OS notifies the script that there is incoming data waiting to be received, ; the following causes a function to be launched to read the data: NotificationMsg := 0x5555 ; An arbitrary message number, but should be greater than 0x1000. OnMessage(NotificationMsg, ReceiveData) Persistent ; Set up the connection to notify this script via message whenever new data has arrived. ; This avoids the need to poll the connection and thus cuts down on resource usage. FD_READ := 1 ; Received when data is available to be read. FD_CLOSE := 32 ; Received when connection has been closed. if DllCall("Ws2_32\WSAAsyncSelect", "UInt", socket, "UInt", A_ScriptHwnd, "UInt", NotificationMsg, "Int", FD_READ|FD_CLOSE) { MsgBox "WSAAsyncSelect() indicated Winsock error " DllCall("Ws2_32\WSAGetLastError") ExitApp } } ConnectToAddress(IPAddress, Port) ; This can connect to most types of TCP servers, not just WinLIRC. ; Returns -1 (INVALID_SOCKET) upon failure or the socket ID upon success. { wsaData := Buffer(400) result := DllCall ("Ws2_32\WSAStartup", "UShort", 0x0002, "Ptr", wsaData) ; Request Winsock 2.0 (0x0002) if result ; Non-zero, which means it failed (most Winsock functions return 0 upon success). { MsgBox "WSAStartup() indicated Winsock error " DllCall("Ws2_32\WSAGetLastError") return -1 } AF_INET := 2 SOCK_STREAM := 1 IPPROTO_TCP := 6 socket := DllCall("Ws2_32\socket", "Int", AF_INET, "Int", SOCK_STREAM, "Int", IPPROTO_TCP) if socket = -1 { MsgBox "socket() indicated Winsock error " DllCall("Ws2_32\WSAGetLastError") return -1 } ; Prepare for connection: SizeOfSocketAddress := 16 SocketAddress := Buffer(SizeOfSocketAddress, 0) NumPut("UShort", 2 ; sin_family , "UShort", DllCall("Ws2_32\htons", "UShort", Port) ; sin_port , "UInt", DllCall("Ws2_32\inet_addr", "AStr", IPAddress) ; sin_addr.s_addr , SocketAddress) ; Attempt connection: if DllCall("Ws2_32\connect", "UInt", socket, "Ptr", SocketAddress, "Int", SizeOfSocketAddress) { MsgBox "connect() indicated Winsock error " DllCall("Ws2_32\WSAGetLastError") ". Is WinLIRC running?" return -1 } return socket ; Indicate success by returning a valid socket ID rather than -1. } ReceiveData(wParam, lParam, *) ; By means of OnMessage, this function has been set up to be called automatically whenever new data ; arrives on the connection. It reads the data from WinLIRC and takes appropriate action depending ; on the contents. { Critical ; Prevents another of the same message from being discarded due to thread-already-running. socket := wParam ReceivedDataSize := 4096 ; Large in case a lot of data gets buffered due to delay in processing previous data. ReceivedData := Buffer(ReceivedDataSize, 0) ReceivedDataLength := DllCall("Ws2_32\recv", "UInt", socket, "Ptr", ReceivedData, "Int", ReceivedDataSize, "Int", 0) if ReceivedDataLength = 0 ; The connection was gracefully closed, probably due to exiting WinLIRC. ExitApp ; The OnExit routine will call WSACleanup() for us. if ReceivedDataLength = -1 { WinsockError := DllCall("Ws2_32\WSAGetLastError") if WinsockError = 10035 ; WSAEWOULDBLOCK, which means "no more data to be read". return 1 if WinsockError != 10054 ; WSAECONNRESET, which happens when WinLIRC closes via system shutdown/logoff. ; Since it's an unexpected error, report it. Also exit to avoid infinite loop. MsgBox "recv() indicated Winsock error " WinsockError ExitApp ; The OnExit routine will call WSACleanup() for us. } ; Otherwise, process the data received. Testing shows that it's possible to get more than one line ; at a time (even for explicitly-sent IR signals), which the following method handles properly. ; Data received from WinLIRC looks like the following example (see the WinLIRC docs for details): ; 0000000000eab154 00 NameOfButton NameOfRemote Loop Parse, StrGet(ReceivedData, "CP0"), "`n", "`r" { if A_LoopField ~= "^(\|BEGIN\|SIGHUP\|END)$" ; Ignore blank lines and WinLIRC's start-up messages. continue ButtonName := "" ; Init to blank in case there are less than 3 fields found below. Loop Parse, A_LoopField, "`s" ; Extract the button name, which is the third field. if A_Index = 3 ButtonName := A_LoopField static PrevButtonName := "", PrevButtonTime := 0, RepeatCount := 0 ; These variables remember their values between calls. if (ButtonName != PrevButtonName || A_TickCount - PrevButtonTime > g_DelayBetweenButtonRepeats) { if HasMethod (Actions.Prototype, ButtonName) ; There is a method associated with this button. Actions.Prototype.%ButtonName%() ; Call the method. else ; Since there is no associated function, briefly display which button was pressed. { if (ButtonName == PrevButtonName) RepeatCount += 1 else RepeatCount := 1 ToolTip "Button from WinLIRC, " ButtonName " (" RepeatCount ")" SetTimer () => ToolTip(), -3000 ; This allows more signals to be processed while displaying the window. } PrevButtonName := ButtonName PrevButtonTime := A_TickCount } } return 1 ; Tell the program that no further processing of this message is needed. } ExitSub(*) ; This function is called automatically when the script exits for any reason. { ; MSDN: "Any sockets open when WSACleanup is called are reset and automatically ; deallocated as if closesocket was called." DllCall("Ws2_32\WSACleanup") } image/svg+xml AutoHotkey modern logo Unfont: fonts converted to paths. Recreation of the Modern logo for AutoHotkey, fonts used are Myriad Pro 12 pt and 36 pt, and Helvetica 16 pt. joedf July 27th 2020 Author unknown for original logo @font-face { font-family:'icons'; /* http://fontastic.me/ */ src:url(fonts/icons.eot?) format("embedded-opentype") } @font-face { font-family:'icons'; src:url (data:font/opentype;base64,AAEAAAANAIAAAwBQRkZUTZJYLx8AAA7QAAAAHEdERUYAKQAaAAAOsAAAAB5PUy8yckiB9AAAAVgAAABWY21hcJAhfusAAAH format("truetype"),url(fonts/icons.woff) format("woff"),url(fonts/icons.svg) format("svg") } #body { height:100%; width:100%; position:fixed; margin:-.72em } [data-content]:before { content:attr(data-content); display:inline-block } #main { height:100%; padding-top:3em; box-sizing:border-box } #head { box-shadow:0px 4px 16px 0px rgba(0,0,0,0.2); z-index:999; background:#3F627F; position:absolute; height:3em; width:100%; -webkit-touch-callout:none; -webkit-user-select:none; -khtml-user-select:none; -moz-user-select:none; -ms-user-select:none; user-select:none; font-family:arial,helvetica,Segoe UI Symbol; color:#eee } #head .skip-nav { clip: rect(1px, 1px, 1px, 1px); height: 1px; overflow: hidden; white-space: nowrap; width: 1px; position: absolute; top: 0; left: 0; background: #000; color: #FFF; z-index: 100;*

```
padding: .4em; } #head .skip-nav:focus { clip: auto; width: auto; height: auto; overflow: auto; } #head div { display:table-cell } #head .h-area { display:table } #head ul { list-style:none; margin:0; padding:0; height:100%; width:100%; display:table; table-layout:fixed } #head .h-tabs { height:100%; width:18em; font-weight:700 } #head .h-tabs ul { position:absolute; top:0; width:inherit } #head .h-tools { font-family:icons; font-size:2em } #head .h-tools.sidebar > ul { max-width:1.75em } #head .h-tools.online > ul { max-width:7em } #head .h-tools.chm > ul { max-width:8.75em } #head .h-tools.main > ul { max-width:3.5em } #head li { display:table-cell; text-align:center; line-height:1.5em; vertical-align:middle; height:100% } #head li > a { display:block; text-decoration:none; color:inherit } :focus:not(:focus-visible) { outline: 0; } #head button, #left button { border-width: 2px; } #head li > button { align-items: normal; background-color: transparent; border: 0; box-sizing: content-box; color: inherit; cursor: pointer; display: block; font: inherit; height: 100%; padding: 0; width: 100%; } #head .h-tools li { max-width:1.75em } #head li:hover { background:#fff; color:#d06d3c; cursor:pointer } @media (hover: none) { #head li:hover { background: inherit; color: inherit; } } #head .selected { background:#fff; color:#3F627F } #head .dropdown { display:none; position:absolute; height:auto; width:1.75em; top:1.5em; z-index:999; box-shadow:0px 8px 16px -4px rgba(0,0,0,0.2) } #head .dropdown li { display:block } #head div.dropdown { background-color:#f5f5f5; color:#000; max-width:30em; position:fixed; white-space:normal; padding:.72em } #head .dropdown li,li.version,li.language { font-family:arial,helvetica,Segoe UI Symbol!important } #head .online,#head .chm { display:none } #left { z-index:999; background:#f9f9f9; color:#000; float:left; width:18em; height:100%; -webkit-touch-callout:none; -webkit-user-select:none; -khtml-user-select:none; -moz-user-select:none; -ms-user-select:none; user-select:none; position:relative } #left.phone { float:none; height:auto; position:absolute; top:3em; bottom:0; left:0; z-index:999 } #left .toc { overflow:auto; -webkit-overflow-scrolling:touch; float:left; width:inherit } #left .load { position:absolute; top:0; right:0; bottom:0; left:0; background-color: inherit; } #left .load div, #right .load div { height:100%; width:100%; display:flex; justify-content:center; align-items:center } .lds-dual-ring { display:inline-block; width:64px; height:64px; opacity: 0; animation: fadeIn .6s ease 1; animation-delay: 2s; animation-fill-mode: forwards; } .lds-dual-ring:after { content:" "; display:block; width:46px; height:46px; margin:1px; border-radius:50%; border:5px solid #fff; border-color:#3F627F transparent #3F627F transparent; animation:rotate 1.2s linear infinite } @keyframes rotate { 0% { transform:rotate(0deg); } 100% { transform:rotate(360deg); } } @keyframes fadeIn { 0% { opacity: 0; } 100% { opacity: 1; } } #left .toc ul, #left .quick .main ul { color:#3F5770; padding:0; margin:0 } #left .toc > ul, #left .quick .main > ul { padding:1em 0 } #left .toc li, #left .quick .main li { list-style:none; white-space:nowrap; overflow:hidden; margin:0; } #left .toc li > span, #left .quick .main li > span { box-sizing:border-box; padding:.1em 0; padding-left:1.5em; display:inline-block; height:100%; width:100%; border-left:.2em solid transparent; cursor: pointer; } #left .toc ul > li > ul { border-left:1px solid silver; margin-left:1.8em } #left .toc ul > li.highlighted > ul { border-left:1px solid #d06d3c } #left .toc li.closed > span,#left .toc li.opened > span { color:#000 } #left .toc li.closed > span:before { content:'+'; width:1em; display:inline-block; margin-left:-1em; font-size:1.1em } #left .toc li.opened > span:before { content:'\2212'; width:1em; display:inline-block; margin-left:-1em; font-size:1.1em } #left .toc a, #left .quick .main a { box-sizing:border-box; color:inherit; display:inline-block; text-decoration: none; /* For IE only */ width: 100%; } #left .toc span:hover a:before, #left .quick .main span:hover a:before { text-decoration: underline; } #left .toc button, #left .quick button { align-items: normal; background-color: transparent; border: 0; box-sizing: content-box; color: inherit; cursor: pointer; font: inherit; height: auto; padding: 0; width: 100%; text-align: left; } #left .toc .selected { font-weight:700; border-left:.2em solid #d06d3c } #left .index,#left .search { position:absolute; top:0; left:0; right:0; bottom:0 } #left .label { position:absolute; top:0; left:.72em; right:.72em } #left .input { position:absolute; top:.72em; left:.72em; right:.72em; height:2em } #left .input input { border:1px solid #ccc; width:100%; height:100%; box-sizing:border-box; color:inherit; background:#fff; padding:.3em .25em; font-family:helvetica,Arial,sans-serif; font-size:1em } #left .input input.match { background-color:#E6FFE6 } #left .input input.mismatch { background-color:#fcc } #left .select { position:absolute; top:3em; left:.72em; right:.72em; height:2em } #left .select select { border:1px solid #ccc; width:100%; height:100%; box-sizing:border-box; color:inherit; background:#fff; font-size:inherit } #left .select select>option { font-size:inherit; color:black } #left .select .empty { color:grey !important } #left .list { position:absolute; left:.72em; right:.72em; background-color:#fff; border:1px solid silver; overflow:auto; -webkit-overflow-scrolling:touch } #left .index .list { top:5.28em; bottom:.72em } #left .search .list { top:3em; bottom:3em } #left .search .checkbox { position:absolute; bottom:.72em; left:.72em; right:.72em; height:2em } #left .search .checkbox input { height:100%; width:1em; margin:0 0 0 .5em; vertical-align:middle; text-align: center } #left .search .checkbox label { border:1px solid #ccc; box-sizing:border-box; padding-left:2em; height:100%; position:absolute; left:0; right:0; bottom:0; cursor: pointer; line-height: 1.9em; } #left .search .checkbox .updown { border:1px solid #ccc; position:absolute; bottom:0; right:0; top:0; width: 1.15em; } #left .search .checkbox .up { position:absolute; left:0; right:0; top:0; height: 50%; } .triangle-up { width:0; margin: auto; position: absolute; left: 0; right: 0; top: 25%; border-style:solid; border-width:0 .4em .4em .4em; border-color:transparent transparent #505050 transparent } #left .search .checkbox .up:hover { background-color:#c2c2c2; cursor:pointer } #left .search .checkbox .down { position:absolute; bottom:0; left:0; right:0; height: 50%; } #left .deprecated { color: brown !important; } .triangle-down { width:0; margin: auto; position: absolute; left: 0; right: 0; bottom: 25%; border-style:solid; border-width:.4em .4em 0 .4em; border-color:#505050 transparent transparent transparent; } #left .search .checkbox .down:hover { background-color:#c2c2c2; cursor:pointer } #left .list > a { text-decoration:none; color:inherit; line-height:1em; padding:.3em .25em; white-space:nowrap; text-overflow:ellipsis; overflow:hidden; display:block; cursor:default } #left .list > a:hover { color:inherit; background-color:silver; cursor:default } #left .list > a.selected { background-color:grey; color:#FFF } #left .tab { visibility: hidden; } #left .tab.full { height: 100%; /* fallback for IE8 */ height: calc(100% - 2em); transition: height .1s; } #left .tab.shrinked { height: 70%; transition: height .1s; } #left .quick { width: 100%; height: calc(30% - 2em); float: left; } #left .quick .main { overflow: auto; -webkit-overflow-scrolling: touch; height: 100%; width: inherit; } #left .quick .main.no-scroll { overflow: hidden; } #left .quick div { float: left; } #left .quick .header { height: 2em; line-height: 2em; cursor:pointer; border-top: 1px solid silver; border-bottom: 1px solid silver; } #left .quick .header .chevron { font-family: icons; font-size: 1.1em; margin: 0 .2em 0 .3em; } #left .quick .header .chevron.down:before { content: '\25BC'; } #left .quick .header .chevron.right:before { content: '\25B6'; } .dragbar { background-color:#eee; width:3px; cursor:col-resize; z-index:1000; position:absolute; top:0; bottom:0; left:18em } .ghostbar { width:3px; background-color:#000; opacity:0.5; position:absolute; cursor:col-resize; z-index:1000; height:100%; top:0 } #right { height:100%; overflow:auto; -webkit-overflow-scrolling:touch; outline:none; } div#right { display: grid; grid-template-columns: 1fr; } #right .load { grid-row-start: 1; grid-column-start: 1; } #frame { border:0; height:100%; width:100%; display:block; position:relative; grid-row-start: 1; grid-column-start: 1; } #frame.hidden { opacity: 0; visibility: hidden; } #frame.visible { opacity: 1.0; visibility: visible; transition: opacity .1s, visibility .1s; } #right .area { padding:.72em } #right .footer { padding:15px 0 15px 0; margin-top:20px; opacity:0.5; border-top:1px solid silver } #right .back-to-top { display:none; line-height:20px; width:40px; font-family:icons; font-size:200%; text-align:center; position:fixed; bottom:10px; right:10px; z-index:999; background-color:#3F627F; color:white; cursor:pointer; padding-top:10px; padding-bottom:10px; opacity:0.5 } div#right .back-to-top { right:27px; } #right .back-to-top:hover { opacity:1.0 } @media (hover: none) { #right .back-to-top:hover { opacity: 0.5; } } #right .back-to-top:before { content:'\25B2' } #right pre.parent { position:relative } #right pre.origin { margin:0; padding:0; background-color:inherit; border:0; line-height:inherit } #right pre.parent > div.buttons { display:none; position:absolute; right:0; top:0 } #right pre.parent > div.buttons > a { background-color:#777; color:#fff; cursor:pointer; display:inline-block; font-size:1.3em; line-height:1em; padding:.165em; font-family:icons } #right pre.parent > div.buttons > a:hover { cursor:pointer; background:rgba(0,0,0,.9); filter:progid:DXImageTransform.Microsoft.gradient (GradientType=0,startColorstr='#000000',endColorstr='#000000'); /* IE8 */ text-decoration:none } #right .headLine:hover:after { content:" \0000B6"; color:#888 } #right .headLine > a.headLink, .ver > a { color:inherit !important } #right .headLine > a.headLink:hover { text-decoration:none; color:inherit } #right .extLink:after { content:"\00A0\00AC"; text-decoration:none; text-decoration-color: transparent; font-family:icons; vertical-align:-2px; } #right .highlight { background-color:#ff9632; color:#000 } #right .highlight.current { background-color:#ff0000; color:#fff } #right table.mobile { border:solid 1px silver; border-collapse:collapse } #right table.mobile tbody { border-top:solid 1px silver } #right table.mobile td { vertical-align:top; padding:.3em .5em; border:none; width:100% } #right table.mobile p { margin-top:0 } #right table.mobile td:first-child { background-color:#F6F6F6; width:auto; white-space:nowrap } #right .deprecated { color: brown; text-decoration: none; border-bottom: 1px dashed brown; } pre[class="origin"] span > a,pre[class="origin"] span > a:hover { color:inherit } code span > a,code span > a:link,code span > a:hover { color:inherit !important } pre > .cmd, pre > .bif, code > .cmd, code > .bif { color:#0148c2 } pre > .met, code > .met { color:#097f9a } pre > .cfs, pre > .dec, code > .cfs, code > .dec { color:#6F008A } pre > .str, code > .str { color:#A31515 } pre .str > .esc, code .str > .esc { color:#FF0000 } pre > .biv, pre > .cls, code > .biv, code > .cls { color:#006400 } pre > .dir, code > .dir { color:green } pre > .lab, pre > .fun, code > .lab, code > .fun { font-weight:bold; color:#290e90 } pre > .num, code > .num { color:#1a6c4e } pre.origin em, code em, pre .cmt, code .cmt { color:#708090 !important } @media print { #head,#left,.dragbar {display:none;} #body {position:static;} #main {padding-top: 0; } } loadJQuery(); addPrototypeMethods(); // --- Get infos about this script file --- var scriptElement = document.querySelector('script [src$="static/content.js"]'); var scriptDir = scriptElement.src.substr(0, scriptElement.src.lastIndexOf('/')); // --- User data --- var user = { fontSize: 1.0, clickTab: 0, displaySidebar: true, colorTheme: 0, collapseQuickRef: false } // --- Cached data --- // To have the data remain while navigating through the docs, it'll be stored into // sessionStorage. Fallbacks to window.name if sessionStorage is not supported. // Note: CHM doesn't support window.localStorage/sessionStorage or cookies. var cache = { colorTheme: user.colorTheme, scriptDir: scriptDir, fontSize: user.fontSize, forceNoFrame: false, forceNoScript: false, clickTab: user.clickTab, displaySidebar: user.displaySidebar, sidebarWidth: '18em', collapseQuickRef: user.collapseQuickRef, RightIsFocused: true, toc_clickItem: null, toc_clickItemTemp: null, toc_scrollPos: 0, index_filter:-1, index_input: "", index_clickItem: 0, index_scrollPos: 0, search_data: {}, search_highlightWords: false, search_input: "", search_clickItem: 0, search_scrollPos: 0, load: function() { if (window.sessionStorage) { var data = JSON.parse(window.sessionStorage.getItem('data')); if (!data) return false; } else try { var data = JSON.parse(window.name); } catch(e) { return false; } if (data.scriptDir != scriptDir) return false; $.extend(this, data); return true; }, save: function() { if (window.sessionStorage) window.sessionStorage.setItem('data', JSON.stringify(this)); else window.name = JSON.stringify(this); }, set: function(prop, value) { this[prop] = value; try { postMessageToFrame('updateCache', [prop, value]) } catch(e) {} return value; } }; // --- Main Execute Area --- // Set global variables: var forceNoScript = forceNoScript || false; var isCacheLoaded = cache.load(); var workingDir = getWorkingDir(); var relPath = location.href.replace(workingDir, ''); var equivPath = $('meta[name|="ahk:equiv"]').prop('content'); var isInsideCHM = (location.href.search(/::/) > 0) ? 1 : 0; var supportsHistory = (history.replaceState) && ! isInsideCHM; var isFrameCapable = !cache.forceNoFrame && (isInsideCHM || supportsHistory); var isInsideFrame = (window.self !== window.top); var isSearchBot = navigator.userAgent.match(/googlebot|bingbot|slurp/i); var isTouch = !!("ontouchstart" in window) || !!(navigator.msMaxTouchPoints); var isOpera = (!!window.opr
```

```
&& !!opr.addons) || !!window.opera || navigator.userAgent.indexOf(' OPR/') >= 0; // Opera 8.0+ var isFirefox = typeof InstallTrigger !== 'undefined'; // Firefox 1.0+ var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0; // At least Safari 3+: "[object HTMLElementConstructor]" var isIE = /*@cc_on!@*/false || !!document.documentMode; // Internet Explorer 6-11 var isIE8 = !-[1,]; // Internet Explorer 8 or below var isEdge = !isIE && !!window.StyleMedia; // Edge 20+ var isChrome = !!window.chrome && (!!window.chrome.webstore || !!window.chrome.csi); // Chrome 1+ var isBlink = (isChrome || isOpera) && !!window.CSS; // Blink engine detection var structure = new ctor_structure; var toc = new ctor_toc; var index = new ctor_index; var search = new ctor_search; var features = new ctor_features; var translate = {dataPath: scriptDir + '/source/data_translate.js'}; var deprecate = {dataPath: scriptDir + '/source/data_deprecate.js'}; scriptElement.insertAdjacentHTML('afterend', structure.metaViewport); var isPhone = (document.documentElement.clientWidth <= 600); (function() { // Exit the script if the user is a search bot. This is done because we want // to prevent the search bot from parsing the elements added via javascript, // otherwise the search results would be distorted: if (isSearchBot) return; // Get user data: if (!isCacheLoaded) { if (isInsideCHM) { var m = scriptDir.match (/mk:@MSITStore:(.*?)\\[^\\]+\.chm/i); if (m[1]) loadScript(decodeURI(m[1]) + '\\chm_config.js', function () { try { $.extend(cache, overwritePropsProps(user, config)); setInitialSettings(); } catch (e) {} }); } else if (window.localStorage) { config = JSON.parse(window.localStorage.getItem('config')); $.extend(cache, overwritePropsProps(user, config)); setInitialSettings(); } else if (navigator.cookieEnabled) { config = document.cookie.match(/config=([^;]+)/); config && (config = JSON.parse(config[1])); $.extend(cache, overwritePropsProps(user, config)); setInitialSettings(); } } else setInitialSettings(); function setInitialSettings() { // font size if (!isFrameCapable && cache.fontSize != 1) $('head').append(' '); // color theme if (cache.colorTheme) structure.setTheme(cache.colorTheme); } // Exit the script on sites which doesn't need the sidebar: if (forceNoScript || cache.forceNoScript) return; // Special treatments for pages inside a frame: if (isFrameCapable) { if (isInsideFrame) { if (cache.fontSize != 1) $('head').append(' '); normalizeParentURL = function() { postMessageToParent('normalizeURL', [$.extend({}, window.location), document.title, supportsHistory ? history.state : null, equivPath]); if (cache.toc_clickItemTemp) if (supportsHistory) history.replaceState($.extend(history.state, {toc_clickItemTemp: cache.toc_clickItemTemp}), null, null); cache.set('toc_clickItemTemp', null); } normalizeParentURL(); $(window).on('hashchange', normalizeParentURL); structure.setTheme(cache.colorTheme); structure.addShortcuts(); structure.addAnchorFlash(); structure.saveCacheBeforeLeaving(); if (!isIE) { structure.hideFrameBeforeLeaving(); postMessageToParent('unhideFrame', []); } $(document).ready(function() { $('html').attr({ id: 'right'}); features.add(); }); $(window).on('message onmessage', function(event) { var data = JSON.parse(event.originalEvent.data); switch(data[0]) { case 'updateCache': if(typeof data[1] === 'object') $.extend(cache, data[1]); else cache[data[1]] = data[2]; break; case 'highlightWords': search.highlightWords(data[1]); break; case 'scrollToMatch': search.scrollToMatch(data[1]); break; case 'setTheme': structure.setTheme(data[1]); break; } }); return; } else { $(window).on('message onmessage', function(event) { var data = JSON.parse(event.originalEvent.data); switch(data[0]) { case 'normalizeURL': var relPath = data[1].href.replace(workingDir, ''); try { if (history.replaceState) history.replaceState(null, null, data[1].href); } catch(e) { if (history.replaceState) history.replaceState(null, null, "?frame=" + encodeURI (relPath).replace(/#/g, '%23')); } document.title = data[2]; if (structure.modifyTools) structure.modifyTools(relPath, data[4]); if ($('#left > div.toc li > span.selected a').attr ('href') == data[1].href) break; else if (data[3] && data[3].toc_clickItemTemp) { toc.deselect($('#left > div.toc')); $('#left > div.toc li > span').eq(data [3].toc_clickItemTemp).trigger('select'); } else toc.preSelect($('#left > div.toc'), data[1]); break; case 'pressKey': structure.pressKey(data[1]); break; case 'updateQuickRef': structure.updateQuickRef(data[1], data[2]); break; case 'hideFrame': $('#right .load').hide().show(0); // reload animation document.getElementById ('frame').className = 'hidden'; break; case 'unhideFrame': document.getElementById('frame').className = 'visible'; break; } }); $(window).on('hashchange', function () { structure.openSite(location.href); }); } } // Add elements for sidebar: structure.build(); // Load current URL into frame: if (isFrameCapable) $(document).ready (function() { if (window.sessionStorage) window.sessionStorage.setItem('data', JSON.stringify(cache)); else document.getElementById('frame').contentWindow.name = JSON.stringify(cache); structure.openSite(scriptDir + '/../' + (getUrlParameter('frame') || relPath)); }); // Modify the site: structure.modify(); if (!isFrameCapable) $(document).ready(features.add); toc.modify(); index.modify(); search.modify(); })(); // --- Constructor: Table of content --- function ctor_toc() { var self = this; self.dataPath = scriptDir + '/source/data_toc.js'; self.create = function(input) { // Create and add TOC items. var ul = document.createElement("ul"); for(var i = 0; i < input.length; i++) { var text = input[i][0]; var path = input[i][1]; var subitems = input[i][2]; if (path != '') { var el = document.createElement("a"); el.href = workingDir + path; if (cache.deprecate_data[path]) el.className = "deprecated"; } else var el = document.createElement("button"); if (isIE8) el.innerHTML = text; else { el.setAttribute("data-content", text); el.setAttribute("aria-label", text); } var span = document.createElement("span"); span.innerHTML = el.outerHTML; var li = document.createElement("li"); li.title = text; if (cache.deprecate_data[path]) li.title += "\n\n" + T("Deprecated. New scripts should use {0} instead.").format (cache.deprecate_data[path]); if (subitems != undefined && subitems.length > 0) { li.className = "closed"; li.innerHTML = span.outerHTML; li.innerHTML += self.create(subitems).outerHTML; } else li.innerHTML = span.outerHTML; ul.innerHTML += li.outerHTML; } return ul; }; // --- Modify the elements of the TOC tab --- self.modify = function() { if (!retrieveData(self.dataPath, "toc_data", "tocData", self.modify)) return; if (!retrieveData(deprecate.dataPath, "deprecate_data", "deprecateData", self.modify)) return; $toc = $('#left div.toc').html(self.create(cache.toc_data)); $tocList = $toc.find('li > span'); // --- Fold items with subitems --- $toc.find('li > ul').hide(); // --- Hook up events --- // Select the item on click: registerEvent($toc, 'click', 'li > span', function() { $this = $(this); cache.set('toc_clickItem', $tocList.index(this)); cache.set('toc_scrollPos', $toc.scrollTop()); // Fold/unfold item with subitems: if ($this.parent().has("ul").length) { $this.siblings("ul").slideToggle (100); $this.closest("li").toggleClass("closed opened"); } // Highlight and open item with link: if ($this.has("a").length) { self.deselect($toc); $this.trigger('select'); structure.openSite($this.children('a').attr('href')); structure.focusContent(); return false; } }); // Highlight the item and parents on select: registerEvent($toc, 'select', 'li > span', function() { $this = $(this); // Highlight the item: $this.addClass("selected"); // Highlight its parents: $this.parent("li").has('ul').addClass('highlighted'); $this.parent().parents('li').addClass('highlighted'); // Unfold parent items: $this.parents("ul").show(); $this.parents("ul").closest("li").removeClass('closed').addClass ('opened'); }); // --- Show scrollbar on mouseover --- if (!isTouch) // if not touch device. { $toc.css("overflow", "hidden").hover(function() { $(this).css("overflow", "auto"); }, function() { $(this).css("overflow", "hidden"); }); } self.preSelect($toc, location); if (!isFrameCapable || cache.search_input) $(document).ready(function() { setTimeout( function() { self.preSelect($toc, location); }, 0); }); }; self.preSelect = function($toc, url) { // Apply stored settings. var tocList = $toc.find('li > span'); var clicked = tocList.eq(cache.toc_clickItem); var found = null; var foundList = []; var foundNoHashList = []; var url_href = (url.href.slice(-1) == '/') ? url.href + 'index.htm' : url.href; for (var i = 0; i < tocList.length; i++) { var href = tocList[i].firstChild.href; if (!href) continue; // Search for items matching the address: if (href == url_href) foundList.push($(tocList[i])); // Search for items matching the address without anchor: else if (href == url_href.substring(0, url_href.length - url.hash.length)) foundNoHashList.push($(tocList[i])); } if (foundList.length) found = $(foundList).map($.fn.toArray); else if (foundNoHashList.length) found = $(foundNoHashList).map($.fn.toArray); var el = found; // If the last clicked item can be found in the matches, use it instead: if (clicked.is(found)) el = clicked; else cache.set('toc_scrollPos', ""); // Force calculated scrolling. // If items are found: if (el) { // Highlight items and parents: self.deselect($toc); el.trigger('select'); // Scroll to the last match: if (cache.toc_scrollPos != "" || cache.toc_scrollPos != 0) $toc.scrollTop(cache.toc_scrollPos); if (!isScrolledIntoView(el, $toc)) { el[el.length-1].scrollIntoView(false); $toc.scrollTop($toc.scrollTop()+100); } } } self.deselect = function($toc) { // Deselect all items. $toc.find("span.selected").removeClass ("selected"); $toc.find(".highlighted").removeClass("highlighted"); } } // --- Constructor: Keyword search --- function ctor_index() { var self = this; self.dataPath = scriptDir + '/source/data_index.js'; self.create = function(input, filter) { // Create and add the index links. var output = ''; input.sort(function(a, b) { var textA = a [0].toLowerCase(), textB = b[0].toLowerCase() return textA.localeCompare(textB); }); for (var i = 0, len = input.length; i < len; i++) { if (filter != -1 && input[i][2] != filter) continue; var a = document.createElement("a"); a.href = workingDir + input[i][1]; a.setAttribute("tabindex", "-1"); if (isIE8) a.innerHTML = input[i][0]; else { a.setAttribute("data-content", input[i][0]); a.setAttribute("aria-label", input[i][0]); } output += a.outerHTML; } return output; }; self.modify = function() { // Modify the elements of the index tab. if (!retrieveData(self.dataPath, "index_data", "indexData", self.modify)) return; var $index = $('#left div.index'); var $indexSelect = $index.find('.select select'); var $indexInput = $index.find('.input input'); var $indexList = $index.find('div.list'); // --- Hook up events --- // Filter list on change: $indexSelect.on('change', function(e) { cache.set('index_filter', this.value); if(this.value == -1) $(this).addClass('empty'); else $(this).removeClass('empty'); $indexList.html(self.create(cache.index_data, this.value)); structure.addEventsForListBoxItems($indexList); }); // Select closest index entry and show color indicator on input: $indexInput.on('keyup input', function(e, noskip) { var $this = $(this); var prevInput = cache.index_input; // defaults to undefined var input = cache.set ('index_input', $this.val().toLowerCase()); // if no input, remove color indicator and return: if (!input) { $this.removeAttr('class'); return; } // Skip subsequent index-matching if we have the same query as the last search, to prevent double execution: if (!noskip && input == prevInput) return; // Otherwise find the first item which matches the input value: var indexListChildren = $indexList.children(); var match = self.findMatch(indexListChildren, input); // Select the found item, scroll to it and add color indicator: if (match.length) { match.click(); // Scroll to 5th item below the match to improve readability: scrollTarget = match.next().next().next().next().next(); if (!scrollTarget.length) { scrollTarget = indexListChildren.last(); }; scrollTarget[0].scrollIntoView(false); $this.attr('class', 'match'); // 'items found' } else $this.attr ('class', 'mismatch'); // 'items not found' }); $indexSelect.val(cache.index_filter).trigger('change'); self.preSelect($indexList, $indexInput); if (!isFrameCapable || cache.index_input) $(document).ready(function() { setTimeout( function() { self.preSelect($indexList, $indexInput); }, 0); }); }; self.findMatch = function (indexListChildren, input) { var match = {}; if (!input) return match; for (var i = 0; i < indexListChildren.length; i++) { var text = isIE8 ? indexListChildren[i].innerText : indexListChildren[i].getAttribute('data-content'); var listitem = text.substr(0, input.length).toLowerCase(); if (listitem == input) { match = indexListChildren.eq(i); break; } } return match; }; self.preSelect = function($indexList, $indexInput) { // Apply stored settings. var clicked = $indexList.children().eq(cache.index_clickItem); $indexInput.val(cache.index_input); if (cache.index_scrollPos == null) $indexInput.trigger('keyup', true); else { $indexList.scrollTop(cache.index_scrollPos); clicked.click(); } }; } // --- Constructor: Full text search --- function ctor_search() { var self = this; self.dataPath = scriptDir + '/source/data_search.js'; self.modify = function() { // Modify the elements of the search tab. if (!retrieveData(self.dataPath, "search_index", "SearchIndex", self.modify)) return; if (!retrieveData (self.dataPath, "search_files", "SearchFiles", self.modify)) return; if (!retrieveData(self.dataPath, "search_titles", "SearchTitles", self.modify)) return; var $search = $('#left div.search'); var $searchList = $search.find('div.list'); var $searchInput = $search.find('.input input'); var $searchCheckBox = $search.find('.checkbox input'); //
```

--- Hook up events --- // Refresh the search list and show color indicator on input: $searchInput.on('keyup input', function(e, noskip) { var $this = $(this); var prevInput = cache.search_input; // defaults to undefined var input = cache.set('search_input', $this.val()); // if no input, empty the search list, remove color indicator and return: if (!input) { $searchList.empty(); $this.removeAttr('class'); return; } // Skip subsequent search if we have the same query as the last search, to prevent double execution: if (! noskip && input == prevInput) return; // Otherwise fill the search list: cache.set('search_data', self.create(input)); $searchList.html(cache.search_data); structure.addEventsForListBoxItems($searchList); // Select the first item and add color indicator: var searchListChildren = $searchList.children(); if (searchListChildren.length) { searchListChildren.first().click(); cache.set('search_clickItem', 0); $this.attr('class', 'match'); // 'items found' } else $this.attr('class', 'mismatch'); // 'items not found' }); self.preSelect($searchList, $searchInput, $searchCheckBox); if (!isFrameCapable) setTimeout(function() { self.preSelect ($searchList, $searchInput, $searchCheckBox); }, 0); }; self.preSelect = function($searchList, $searchInput, $searchCheckBox) { // Apply stored settings. $searchInput.val(cache.search_input); if (cache.search_scrollPos == null) $searchInput.trigger('keyup', true); else { $searchList.html(cache.search_data); structure.addEventsForListBoxItems($searchList); $searchList.scrollTop(cache.search_scrollPos); $searchList.children().eq(cache.search_clickItem).click(); } $searchCheckBox.prop('checked', cache.search_highlightWords); }; self.convertToArray = function(SearchText) { // Convert text to array. // Normalize whitespace: SearchText = SearchText.toLowerCase().replace(/^ +| +$| +(?=)|\+/, ''); if (SearchText == '') return ''; else // split and remove undefined or empty strings return $(SearchText.split(' ')).filter(function(){return (!!this)}); } self.highlightWords = function(words) { var content = $(isInsideFrame ? 'body' : '#right .area'); if(words) { var qry = self.convertToArray(words); for (var i = 0; i < qry.length; i++) { content.highlight(qry[i]); } self.scrollToMatch(); // Scroll to first match. } else content.removeHighlight(); } self.scrollToMatch = function(direction) { var matches = $(isInsideFrame ? 'body' : '#right .area').find('span.highlight'); if (! matches.length) return; var currMatch = matches.filter('.current'); if (currMatch.length) { index = matches.index(currMatch) + (direction == 'next' ? 1 : -1); if (index > matches.length - 1) index = 0; else if (index < 0) index = matches.length - 1; currMatch.removeClass('current'); } else index = 0; matches.eq(index).addClass('current'); matches.eq(index)[0].scrollIntoView(isIE8 ? true : {block: 'center'}); } self.create = function(qry) { // Create search list. var PartialIndex = {}; var RESULT_LIMIT = 50; qry = self.convertToArray(qry); if (qry == '') return function file_has_all_words(file_index, words, start) { for (; start < words.length; ++start) { var iw = index_partial (words[start]) if (!iw || iw.indexOf(file_index) == -1) return false } return true } // Get each word from index and clone for modification below: var all_results = [] for (var i = 0; i < qry.length; ++i) { var t = qry[i].replace(/(\(|\(\))$/,'') // special case for page names ending with () var w = index_partial(t) w = w ? w.slice() : [] all_results[i] = get_results(w) } var ranked = rank_results(all_results,qry) return append_results(ranked); // Get normal results for each term: function get_results(w) { var c = 0 var ret = [] for (var i = 0; i < w.length && c < RESULT_LIMIT; ++i) { if (!file_has_all_words(w[i], qry, 1)) continue // Skip files which don't have all the words. c++ var f = cache.search_files[w[i]] // data.files excludes '.htm' to save space, so add it back: if (f.indexOf('#') != -1) f = f.replace('#', '.htm#') else f = f + '.htm' // var ret_i = { u: location.href.replace(/\/docs\/./, "/docs/" + f), var ret_i = { u: f, t: (cache.search_titles[w[i]] || f) } ret.push(ret_i) } return ret } function rank_results(aro,terms) { // Organize the info: var aro_k = [] var aro_ua = [] var aro_ka = [] for (var i = 0; i < aro.length; ++i) { aro_k[i] = [] if (aro[i] === undefined) { continue; } for (var j = 0; j < aro[i].length; ++j) { aro_k[i].push(aro[i][j].t) aro_ka.push(aro[i][j].t) aro_ua[aro[i][j].t] = aro[i][j].u } } // Assemble list of unique results: var ukeys = [] for (var i = 0; i < aro_ka.length; ++i) if (ukeys.indexOf(aro_ka[i]) == -1) ukeys.push(aro_ka[i]) // The lower the rank the better // normal ranking (based on page contents): var uranks = [] for (var i = 0; i < ukeys.length; ++i) { uranks[ukeys[i]] = [] for (var j = 0; j < aro_k.length; ++j) { uranks[ukeys[i]][j] = ukeys[i].indexOf(aro_k[j]) } } // Added ranking (based on page names) // and calculate the ranks: for (var i = 0; i < ukeys.length; ++i){ var name = ukeys[i] var tmp = uranks[name] // If the name contains any of the search terms: if (anyContains(name.toLowerCase(),terms)) { tmp.push(0) // Give it a better rank. } else { tmp.push(1,8) // Give it a worse rank, Tweakable! } uranks [name] = array_avg(tmp) } // Sort results by rank average and finalize: var ret = [] for (var i = 0; i < ukeys.length; ++i){ var name = ukeys[i] var url = aro_ua[name] var avg = uranks[name] ret.push({n:name,u:url,a:avg}) } ret.sort(function(a,b){return (a.a - b.a)}) ret.slice(0,RESULT_LIMIT) return ret } function array_avg(arr) { var sum = 0 var total = arr.length for (var j = 0; j < arr.length; ++j){ if ((arr[j] < 0) || (arr[j-1] == arr[j])) total -= 1 // Give a worse rank, duplicate ranks are ignored. else sum += arr[j] } return (sum/total) } function anyContains(str, arr) { // http://stackoverflow.com/a/5582640/883015 for (var i = 0, len = arr.length; i < len; ++i) { if (str.indexOf(arr [i]) != -1) return 1 } return 0 } function append_results(ro) { var output = ''; for (var t = 0; t < ro.length && t < RESULT_LIMIT; ++t) { var a = document.createElement ("a"); a.href = workingDir + ro[t].u; a.setAttribute("tabindex", "-1"); if (isIE8) a.innerHTML = ro[t].n; else { a.setAttribute("data-content", ro[t].n); a.setAttribute ("aria-label", ro[t].n); } output += a.outerHTML; } return output; } function decode_numbers(a) { // Decode a string of [a-zA-Z] based 'numbers' var n = [] for (i = 0; i < a.length; i += 2) n.push(decode_number(a.substr(i, 2))) return n } function decode_number(a) { // Decode a number encoded by encode_number() in build_search.ahk: var n = 0, c for (var i = 0; i < a.length; ++i) { c = a.charCodeAt(i) n = n*52 + c - ((c >= 97 && c <= 122) ? 97 : 39) } return n } function index_whole(word) { // Return a word from the index, decoding the list of files // if it hasn't been decoded already: var files = cache.search_index[word] if (typeof(files) == "string") { files = decode_numbers(files) cache.search_index[word] = files } return files } function index_partial(word) { if (word[0] == '+') // + prefix disables partial matching. return index_whole(word.substr(1)) // Check if we've already indexed this partial word. var files = PartialIndex[word] if (files !== undefined) return files // Find all words in search.index which *contain* this word // and cache the result. files = [] var files_low = [] for (iw in cache.search_index) { var p = iw.indexOf(word) if (p != -1) files.push.apply(p == 0 ? files : files_low, index_whole(iw)) } files = files.concat(files_low) var unique = [] for (var i = 0; i < files.length; ++i) if (unique.indexOf(files[i]) == -1) unique.push(files[i]) PartialIndex[word] = unique return unique } }; } // --- Constructor: Navigation structure --- function ctor_structure() { var self = this; self.metaViewport = ' '; self.template = '*

' + '

-
-
-
- Ξ
- A
-
  -
  -
  -
  -
  -
  -
-
  -
  -
-
- ◄
- ►
-
-
- A¬
-
- S

' + '

Filter

☐ ***Highlight keywords***
'+(isIE8?'':'

')+'
'+(isIE?'':'

')+''+(isFrameCapable?'