HotString Helper 2.0 and AutoCorrect for AHK v2

kunkel321 | User manual | Updated February 15, 2024

Welcome

My hope is that an overview of hotstrings will help explain why HotString Helper 2.0 and the AutoCorrect f() Function, as well as the many word part entries, are made the way they are. Whether good or bad; this manual reads more like a series of journal entries than a traditional software user manual. My advance apologies for the topic jumping and the tangents. I recommend reading the sections in order, since they build on each other.

Brief Overview of HotStrings.

As its name implies, a "hot string" is a predefined sequence of keys that, when pressed in order, will trigger some action by the software. For the purposes of this article, we are referring to *auto-replace* hotstrings, in which the triggering characters are automatically removed. A hotstring mustn't be confused with a "hotkey combination," where multiple keys are pressed once, simultaneously. With hotstrings, the software—in this case, *AutoHotkey*—watches for the string. When the string is typed by a user, the software detects the input sequence, then simulates another, associated, series of key presses which are the pre-defined expansion, or "replacement" text. AutoHotkey is unique in the PC world, partly because of its ease of use, with regard to creating custom hotstrings. Indeed, the desire to allow hotstrings to be constructed in a single line of code was a big part of the impetus for the original creation of AutoHotkey (AHK). For interested readers who are new to hotstrings, a good place to start learning, is the AHK HotStrings Documentation. Intermediate AHK users will know that there is a Hotstring Function that programmatically creates hotstrings at run time. The below discussion covers the first time of hotstrings, not the use of the Hotstring Function.

Uses of HotStrings.

In AHK, a hotstring might be used to activate a block of code, but a more-frequent use is to auto-replace text. "Auto-replacement" hotstrings typically have at least two components: Trigger and Replacement. An example of the syntax, from the AHK docs, is:

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

The AutoHotkey application knows that this is a hotstring definition because of the two sets of double-colons. While running this code, type "*btw*" and upon pressing an "*End Character*" such as space, period, enter, etc., it will be replaced with "by the way." To add more utility, AutoHotkey allows various options that affect how the hotstring works. The options must appear between the first set of colons. Additionally, AutoHotkey allows *in-line comments* to be added at the end of individual lines of code. Comments always begin with a space, then semicolon, and are ignored by the AHK application. They are only used for human consumption. Here is another example:

```
:*:fwiw::for what it's worth ; the asterisk means that no end char is needed.
```

Making a Boilerplate Text Expansion

When AHK users create their own hotstrings, they usually create boilerplate "template" items. Examples include the two above (btw and fwiw). Another example is:

```
:*:sig/::Stephen Kunkel`nAutoHotkey Enthusiast`n555-1234
```

Upon typing "sig/" the software would replace it with:

```
Stephen Kunkel
AutoHotkey Enthusiast
555-1234
```

Please note that the trigger string is suffixed with a slash. A suffix is not the same thing as an *Ending Character*. Indeed, no end char is needed here, because of the asterisk option. It is important for the suffix "/" to be there though, otherwise you couldn't type words beginning in "sig" because it would trigger an unwanted replacement. Having a trigger string prefix would also work (e.g. */sig*, *.sig*, or *;sig*). Rather than using part of a word as the trigger, a user might also choose an *acronym* (technically an "*initialism*" here) made from the first letter of each word (e.g. skae).

Another relevant feature is that multi-line expansions like the one above can be made via *Continuation Section* such as:

```
::skae::
(
Stephen Kunkel
AutoHotkey Enthusiast
555-1234
)
```

This adds more lines to your script file, but it has the effect of being more "*what you see is what you get*," which is nice. As we will see below, HotString Helper 2.0 makes multiline hotstrings, by default, using Continuation Section syntax.

AutoCorrect

Another common use of AutoHotkey auto-replace hotstrings is to have personal *autocorrect libraries*. Long-time users of Microsoft Word will be familiar with Word's AutoCorrect feature which has a library of common typos and misspellings. When you type one of them, Word instantly fixes the error. This is not the same as *SpellCheck*, which locates your misspellings and suggests corrections. AutoCorrect only works *as you type* and it makes changes without prompting the user. An AHK script file with a library of autocorrect items works essentially the same way—but it is not limited to MS Office applications. It works most anywhere that you type text on your PC.

A fundamental difference between boilerplate hotstrings and autocorrect hotstrings is that you *intentionally* place boilerplate items in your documents, emails, etc. You don't intentionally misspell or mistype words or phrases. AutoCorrect items are preemptively added to your system as a safety net, or failsafe. This difference becomes relevant below, when we discuss the f() function and logging.

Many AHK users will already be familiar with the most excellent 2007 AutoCorrect.ahk script by Jim Biancolo. The list, that is written in AutoHotkey v1 code, was largely derived from Wikipedia's list of common misspellings. An updated, for AHK v2, version of AutoCorrect.ahk, by the current writer (kunkel321) added about 1200 "common grammar errors," also from Wikipedia.

Philosophy of best multi-word matches.

What makes a good autocorrect item? Certainly, it makes sense to have corrections for words that are *high-frequency words* in the given language, English for our purposes here. Secondly, having corrections for likely-to-occur misspellings or mis-typings makes sense. As mentioned above, the ubiquitous 2007 AutoCorrect.ahk was based primarily on Wikipedia's lists of common errors. But where does that list come from? Of course, it's the Wikipedia Typo Team! Overall, they do a fantastic job, but let's face it: With 6.7 million articles, written using 4.3 billion words, the Typo Team can't be physically proofreading all of those. My own hypothesis is that different versions of Wikipedia are programmatically compared, and when a particular word is replaced with another similar particular word, X number of times, the different versions of the word are saved and flagged as a "*frequently occurring*" typo-and-correction-pair. This is a solid approach, but I think that sometimes a person writes some really long article on an obscure topic, then someone else comes along and changes the spelling of some word that occurs X times, but *only occurs in said article*. The obscure corrected word then becomes a "frequently misspelled" one. That might explain some of the seemingly obscure typo-correction pairs in the list, such as: *alsation->Alsatian, bernouilli->Bernoulli, lotharingen->Lothringen, lukid->likud, mythraic->Mithraic, papanicalou->Papanicolaou, and valetta->Valletta,* to name a few.

Whatever the reason for the odd entries in AutoCorrect 2007, most of the words are good useful ones. It is noteworthy that the list contains 4637 whole-word corrections. Many of those words have common root words but differ in prefixes (such as *"un-" or "anti-"* or *"re-")* or suffixes (such as *"-ly" or "-ing"* or *"-ness").* When working with simple basic AHK hotstrings (those without hotstring options), it is indeed necessary to use whole words. And so, including the various permutations (single/plural, etc.) for these items is needed. For example, consider this item:

```
::mrak::mark
```

Typing "mrak" results in "mark." But typing "unmrak" does not result in "unmark," because by default, hotstring triggers can't be preceded by alphanumeric characters. Similarly, "mraked" doesn't get corrected, because an End Char needs to be typed to trigger the replacement. There are however, hotstring options for both of these scenarios.

```
:?:mrak::mark ; This word-end item corrects mrak and also unmrak.
:*:mrak::mark ; This word-beginning item corrects mrak and also mraked.
```

They can be combined as well.

```
:*?:mrak::mark
```

This is a "word middle" match and fixes "birthmark," "Denmark," "earmarked," and 172 other words. Of course, not all of the 175 potential fixes are high-frequency words. I mean, how often will a person type the word "supermarketeer?" Still, 175 potential matches for a single autocorrect entry is pretty impressive, in terms of being high utility.

Jim recognized this potential when making the original 2007 AutoCorrect.ahk script. The script contains 83 word beginning items, that use the :*: option, 15 word ending items that use :?:, and two word middles with both options :?*: (seen below).

```
:?*:compatab::compatib  ; Covers incompat* and compat*
:?*:catagor::categor  ; Covers subcatagories and catagories.
```

I had never really expanded on this concept, until being inspired by *Jack Dunning's* book, Beginning AutoHotkey Hotstrings: A Practical Guide for Creative AutoCorrection, Text Expansion and Text Replacement. In chapter three he introduces the reader to using a site that looks up words via wildcard matches, to see how many matches there are to various word parts. For example, referring to these two items:

```
:*?:ugth::ught
:*?:igth::ight
```

Jack writes, "*According to the word lookup site referenced above, the first line of code protects against the misspelling of 133 words through the accidental swapping of the ht. The second line adds another 887 word variations to the same AutoCorrect app.*" As pointed out by Jack (and as logically concluded), the key here; the way to make the best multi-match high utility autocorrect items, is to identify likely misspellings that correspond to a large number of words, but that also don't inadvertently misspell other words.

In 2021 I began the process of manually assessing and, where relevant, converting all 4637 whole word hotstrings from the 2007 AutoCorrect list. I already had the excellent application, WordWebPro installed. WordWeb has a utility for doing wildcard searches, so I used that tool, manually pasting in each word, then progressively deleting and backspacing letter-by-letter from the beginning, then the end of each word, to see which word middles were good multi-match items. An important part of the process was to also trim off the same letters from the beginning/ending of the (misspelled/typo) hotstring trigger string, to ensure that the new, shortened, trigger didn't correspond to—and therefore misspell—any other words. This was a mind-numbingly tedious and time-consuming process. I eventually became sufficiently motivated to create the Word Analyzer Gui (WAG) Thing. As indicated in the WAG forum post,

"The WAG tool won't convert the hotstrings for you, but it will let you simultaneously winnow-down the trigger string and the hotstring and compare each to a list of English words. The winnowing process usually involves trimming the prefix or suffix from a root word (assuming that the typo is in the root). It doesn't "intelligently" remove the same letter from each string, it just removes a letter-at-a-time".

The resulting multi-match autocorrect entries—at least the good ones—are *high-utility items*. The WAG tool itself, however, is not a *high-utility tool*. Really, its only reason for existence is to process autocorrect items. It is a "one-trick pony" in that regard. It served its purpose well, allowing me to process all of the remaining whole word items from the AutoCorrect 2007 list, perhaps 10 or 20 times faster than without it. After completing the list, for a short time, the WAG Thing fell into disuse, but as I later found more whole-word autocorrect items via internet search, ChatGPT, and personal poor typing experiences, I found that the tool was still useful from time to time. Often, when typing, and I misspell or mistype a word, I'll quickly save it as an autocorrect entry. Then later, when I have time, I'll go back to and examine the entry to see if it is a good candidate for a multi-match word part entry (see below section about capturing an existing hotstring entry -- bottom of page 12).

As discussed below, I integrated the tool as a *sub-component of HotString Helper*. I got rid of the "WAG" name, because…. Well, because it's rather silly to name something "Wag" unless it has to do with happy dogs. I guess we'll call it the "*Exam Pane*."

As an example of how the "word-analyzing/exam" process is occasionally useful, while writing this article I used the word "combination." Being a person who is bad at spelling, I typed it "combonation." I can image myself potentially misspelling that word the same way in the future, so I decided it is probably a good candidate for an autocorrect item. The item, written for AHK (v1 or v2) would be:

```
:: combonation::combination
```

By simply adding the options, ":*?:"

```
:*?: combonation::combination
```

This becomes a multi-match item, and matches, "combination, combinational, combinations, recombination, and recombinations." If we trim five characters off of the end, we get,

```
:*?: combon::combin
```

which potentially fixes 40 different words, according to the word list I was using at the time. Note that you can't remove any letters from the beginning,

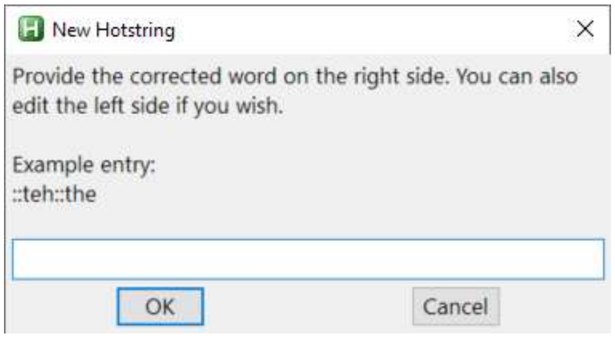
```
:*?: ombon::ombin
```

or it will change "trombone" to the erroneous "trombine." So that's no good.

Let's take a look at the HotString Helper tool, including this "combonation" example, then we'll talk more about the f() Function and the AutoCorrect for v2 script.

Original HotString Helper

On his blog, Jim B. credits Tara Gibb with the original HotString Helper tool, though the AHK Documentation suggests it was Andreas Borutta. While AutoCorrect.ahk 2007 is running, press the Win+H hotkey, and you'll get the image to the right.


New Hotstring

Provide the corrected word on the right side. You can also edit the left side if you wish.

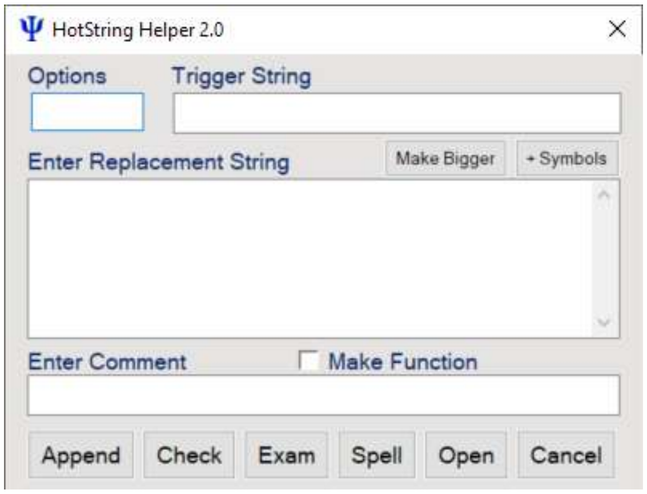
Example entry:
::teh::the

Enter your hotstring and press OK, and the new hotstring gets added to your AutoCorrect file (assuming you haven't compiled it to an exe file). Then the file is saved and reloaded so that your new hotstring is loaded into your PC's RAM and is ready to use. Nice. The version of HotString Helper from 2007 is written in AHK v1, but a v2 remake is available as well.

It's noteworthy that -- as discussed on page two -- AHK has a Hotstring Function which makes a hotstring immediately available to a script. The version 2 remake of the classis HotString Helper does use the Hotstring Function, but HotString Helper 2.0 does not. The ::trigger::replacement style hotstrings only become available when the containing script is reloaded.

HotString Helper 2.0

My own version of HotString Helper works similarly to the classic version, but has several extra features. It's noteworthy that the *"2.0" nomenclature* is not really a version number, per se. It's mostly there to differentiate this version from Tara's original. I chose "2.0" because it is written in AutoHotkey v2. The original motivation for this version was to facilitate the creation of multi-line Continuation Section boilerplate items***, in addition to*** being able to create normal single word autocorrect entries. The first version of this was written in AHK v1 and was called "*HotString Helper – Mult-Line*." HH-ML was then remade with AHK v2 code. Upon combining the above-mentioned WAG Thing components, I considered, "*HotString Helper – Word-Analyzing Multi-Line*," but that is too long, and "*WAML*" is too confusing. So "2.0" it is. It appears as seen in the screenshot on the right. Define your own GUI form colors and icon near the top of the code.



The first obvious difference with HotString Helper 2.0 (hh2), is that each component of the hotstring has its own edit box.

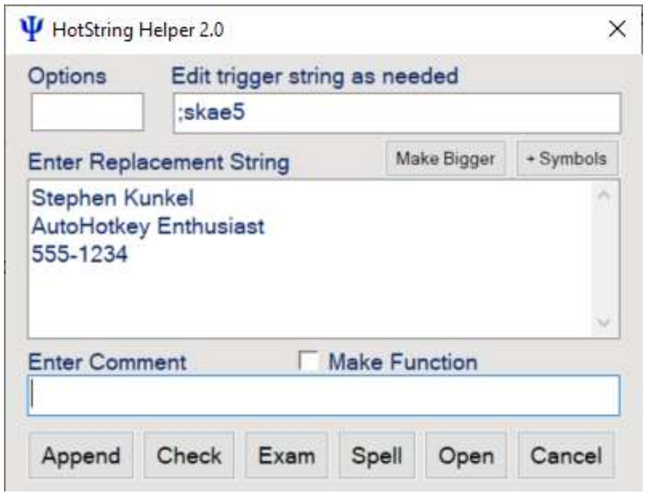
- Options (which are optional)
- Trigger String (a.k.a. hotstring; mandatory)
- Replacement String (a.k.a. expansion text, boilerplate text; mandatory)
- Comment (optional)

Boilerplate Entries

My fake signature here is from the above example.

Stephen Kunkel
AutoHotkey Enthusiast
555-1234

If I select the three lines of text, then press the default hotkey: Win+H (technically, it's "Win+h"), I get what is shown in the image to the right. ---->



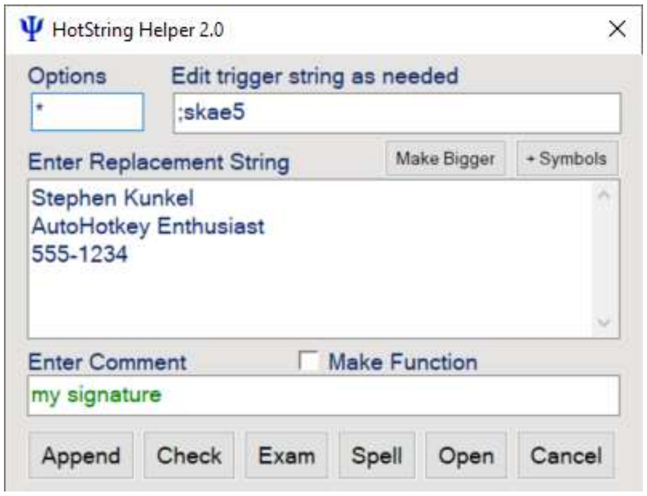
;skae5

Stephen Kunkel
AutoHotkey Enthusiast
555-1234

Notice the automatically-generated trigger string, "**;skae5**." This is an acronym (technically an "initialism") made from the first letter of each word. It's prefixed with a semicolon because I always start my boilerplate triggers with a semicolon. The semicolon potentially confuses things because semicolons delineate in-line comments in AHK code. It's just so convenient though… It's one of the eight Home Keys! If you have a different preferred prefix, define it with the *myPrefix* variable – see below. When creating the acronym, hh2 is coded to use the first character, of the first X words, and to skip words that are Y letters or shorter. Near the top of the code are the variables where X and Y are defined. They are shown below as well.

```
;===Change=options=for=MULTI=word=entry=options=and=trigger=strings=as=desired==
; These are the defaults for "acronym" based boiler plate template trigger strings.
DefaultBoilerPlateOpts := ""  ; PreEnter these multi-word hotstring options; "*" = end char not needed, etc.
myPrefix := ";"               ; Optional character that you want suggested at the beginning of each hotstring.
addFirstLetters := 5          ; Add first letter of this many words. (5 recommended; 0 = don't use feature.)
tooSmallLen := 2              ; Only first letters from words longer than this. (Moot if addFirstLetters = 0)
mySuffix := ""                ; An empty string "" means don't use feature.
```

For the purposes of this demonstration, I manually added an asterisk to the Options edit box. This causes the hotstring to activate without the need of an End Char. Therefore, the moment I press the '5' it will trigger the replacement text. I also added a comment, again for demonstration purposes. The image to the right shows the hh2 form with these things added.



*

;skae5

Stephen Kunkel
AutoHotkey Enthusiast
555-1234

my signature

Upon clicking the Append button, the following is added to the bottom of the AutoCorrect.ahk file:

```
:*:;skae5:: ; my signature
(
Stephen Kunkel
AutoHotkey Enthusiast
555-1234
)
```

Notes:

-Pressing the *Enter key* will also append the item. However, pressing the Enter key will not have this effect if you are editing the replacement text. When editing the replacement text, the Enter key merely types an {Enter}.

-Note also, that holding the *Shift Key* while clicking the Append button (i.e. *Shift+Click*) will save the hotstring to the Windows clipboard instead of appending it to the AutoCorrect file. In this situation, if the Validation Warning appears, you have to keep holding the Shift key for the item to be placed on the clipboard. Note also, the item will only be on the clipboard while the hh2 form is shown. Once you close the form, then the previous Windows clipboard contents will be restored.

Sometimes I like to have Tab characters to the right of the multiline items. For example:
Reading →→
Writing →→
Math →→→

This allows me to easily add (for example) the reading, writing, and math test scores into their own column to the right of the names, after expanding the boilerplate text. When you select the multiline item, and press Win+H, *if a Tab character is detected at the end of the replacement string, then the command to keep the white space is added to the hotstring*, such as:

```
::;rwm::
(RTrim0
 reading
 writing
 math
)
```

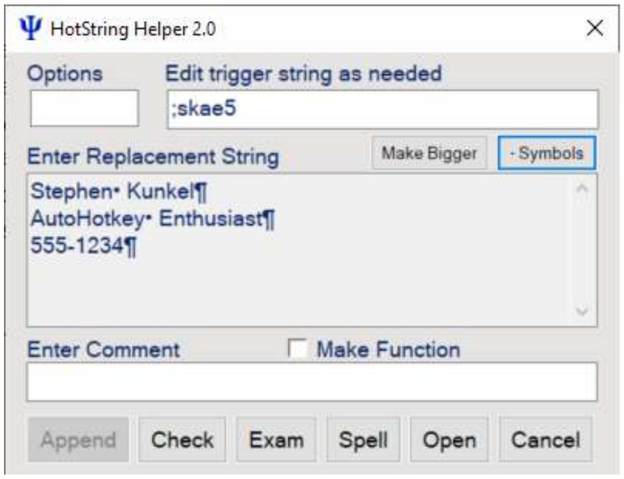
Without the *RTrim0* (Right Trim Zero, where zero means 'off'), AHK would remove the last Tab characters when it expands the text.

Show Symbols

Please note the two small buttons just above the Replacement String edit box. The **+/- Symbols** button toggles the visibility of a *Pilcrow, space dot, and right arrow* (for Enter, Space, Tab, respectively) as in the images below. When the symbols are visible, the Replacement String box becomes *Read-Only*, and the Append button becomes inactive. The actual symbols for these can be changed near the top of the code. The settings are:

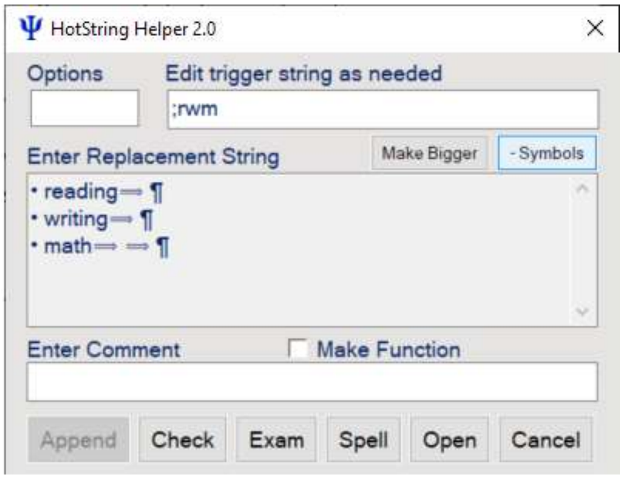
```
;====Assign=symbols=for="Show Symb"=button=====================================
myPilcrow := "¶"   ; Okay to change symbols if desired.
myDot := "• "      ; adding a space (optional) allows more natural wrapping.
myTab := "⟹ "      ; adding a space (optional) allows more natural wrapping.
```

Be sure to have your .ahk file formatted as *UTF-8 with BOM* or the symbols might not get displayed correctly.



;skae5

Stephen• Kunkel¶
AutoHotkey• Enthusiast¶
555-1234¶



;rwm

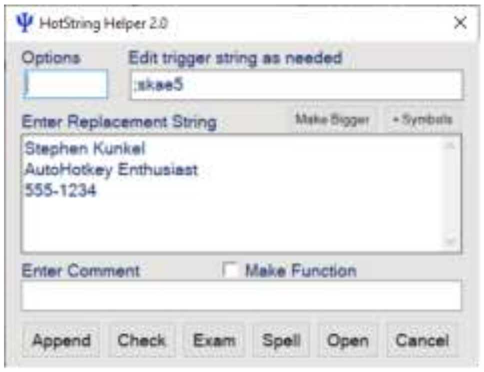
• reading⟹ ¶
• writing⟹ ¶
• math⟹ ⟹ ¶

Special thanks to user "off," who helped me work out how to create this effect.

Make Replacement Box Bigger

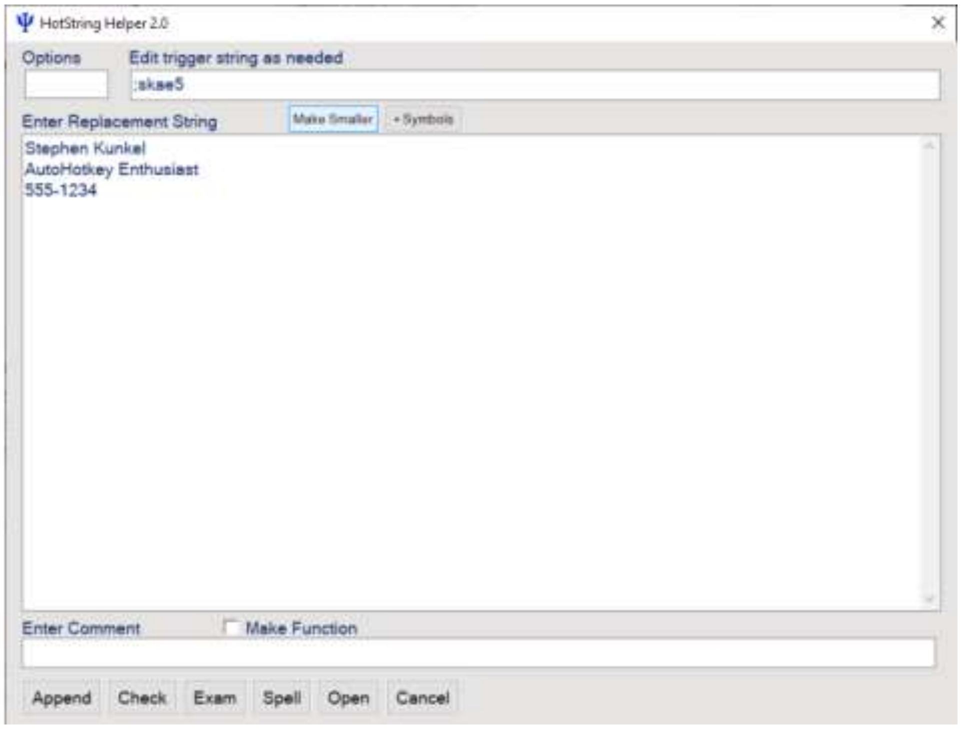
The **Make Bigger/Smaller** button toggles the size of the Replacement String box, as seen in the images below. The amount of increase can be customized with the following variables.

```
; ======Change=size=of=GUI=when="Make Bigger"=is=invoked=======================
HeightSizeIncrease := 300 ; Numbers, not 'strings,' so no quotation marks.
WidthSizeIncrease := 400
```



:skae5

Stephen Kunkel
AutoHotkey Enthusiast
555-1234

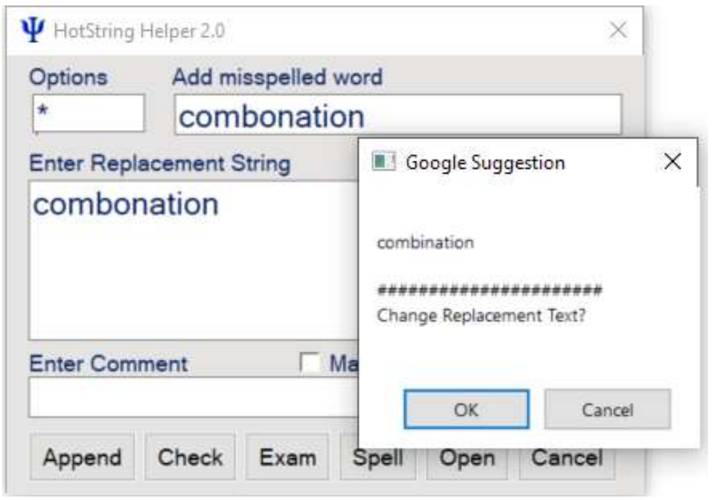


:skae5

Stephen Kunkel
AutoHotkey Enthusiast
555-1234

AutoCorrect word entries

As previously indicated, the above multiline boilerplate template features are a big part of the hh2 tool. The *other* big part is the AutoCorrect multi-match entry tool kit. Typically, the hh2 tool is activated by first selecting a bit of text, then pressing the hotkey (Win+H by default). The tool attempts to determine if you are creating a *boilerplate item*, -or- an *autocorrect item*. There is also a *check-for-existing-hotstring* mechanism, discussed below. The check for template vs. autocorrect is as follows: If the selected text contains three or more space " " characters, *or* if it contains any new line "`n" characters, then assume it is a boilerplate template item and extract the initials from the first X words, of Y length. Also, if it is a boilerplate template item, use the settings defined in the variables, "*DefaultBoilerPlateOpts, myPrefix*, and *mySuffix*." If there is text selected that doesn't match this, then assume an autocorrect entry, and use the setting defined in the variable just below those, called, "*DefaultAutoCorrectOpts*." If no text is selected, just open a blank hh2 form. The code is commented with further explanations of what these variables are for.



combination

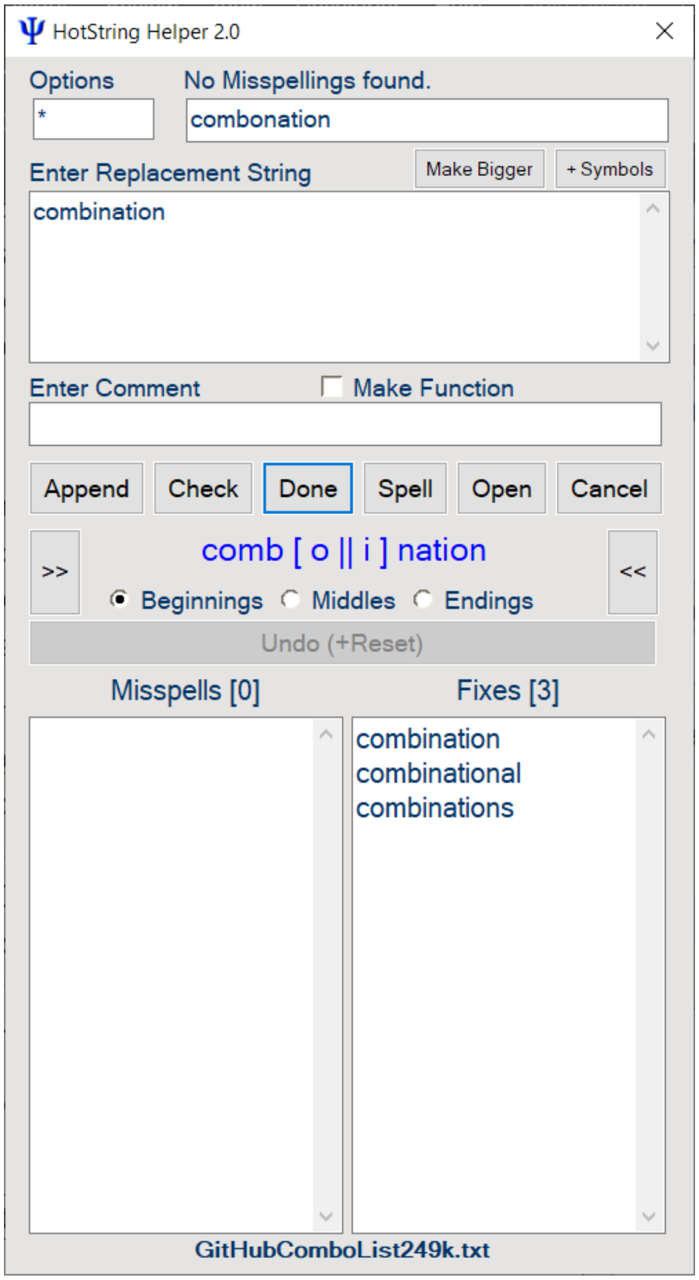
######################
Change Replacement Text?

With the example screenshot on the right, I mis-typed "*combination*." It occurred to me that this might be a good candidate for an autocorrect item, so I selected it with my mouse and activated hh2 via Win+H. In order to get the correct spelling for the replacement box, I then clicked the Spell button. The *Spell* tool is not a "spell checker" per se. It sends the word to Google Search, then returns Google's "*Did you mean….*" response. Google doesn't always provide a suggestion, but it usually does. The suggestion appears in a Message Box, as shown in the image. Clicking OK will update the text that is in the Replacement String edit box.

Note: My old eyes have trouble seeing the text in the form, so I added a Large Font toggle. In the image the font is toggled to be larger. Please do not confuse the function of the "Make Bigger" button, with changing font size. The font is changed in one of two ways: (1) Ctrl+Up Arrow/Ctrl+Down Arrow or (2) Ctrl+Mouse Wheel Up/Down. The default smaller size is s11, and the larger size is s15. This is not a "zoom," it's a toggle.

Word Analysis

The word “*Analyze*” is too big to fit on the button. I thought about having it say “*Anal*,” but ultimately, I decided on “*Exam*.” Click the Exam button to examine the word. When you do, the bottom part of the form, the *Exam Pane*, “pops down” and the Exam button text changes to “*Done*” as seen on the right.



HotString Helper 2.0

Options: *

No Misspellings found.

combonation

Enter Replacement String

combination

Enter Comment

comb [o || i] nation

[✓ Beginnings] Middles Endings

Misspells [0]

Fixes [3]

combination
combinational
combinations

GitHubComboList249k.txt

As suggested in the above section, “*Philosophy of best multi-word matches*,” the main goal here is to simultaneously “winnow down” the trigger and the replacement words, while also experimenting with word -beginnings, -endings, and -middles, to figure out what combination gives us the most “bang for our buck” in terms of being a high-utility multi-match autocorrect entry.

When the Exam Pane of the hh2 form is expanded, the word-beginning/ending/middle options from the Options box will be propagated to the Exam Pane radio buttons. Changing the radio buttons will update the Options box. Tip: Double- (triple-?) clicking the middle radio button will set it to false/unselected, and the * and ? will both be removed from the Options box.

With my default setting of asterisk, you can see that there are already three potential words that would be corrected with the hotstring

```
:*:combonation::combination
```

Trimming

The next step will be to progressively trim-off letters from the beginning and/or ending, using the **[>>]** and **[<<]** buttons. Clicking the wide Undo button (or pressing Ctrl+Z) will undo the trims until you are back to the original word. Shift+Click (Shift+Ctrl+Z) will undo all the trims (i.e. Reset the word).

Delta String.

Please note the *blue delta string:* **comb [o | | i] nation.** This serves the purpose of giving us a quick visual of how many characters can be trimmed from the beginning and end before the “typo part” of the trigger is removed. The delta string has four components:

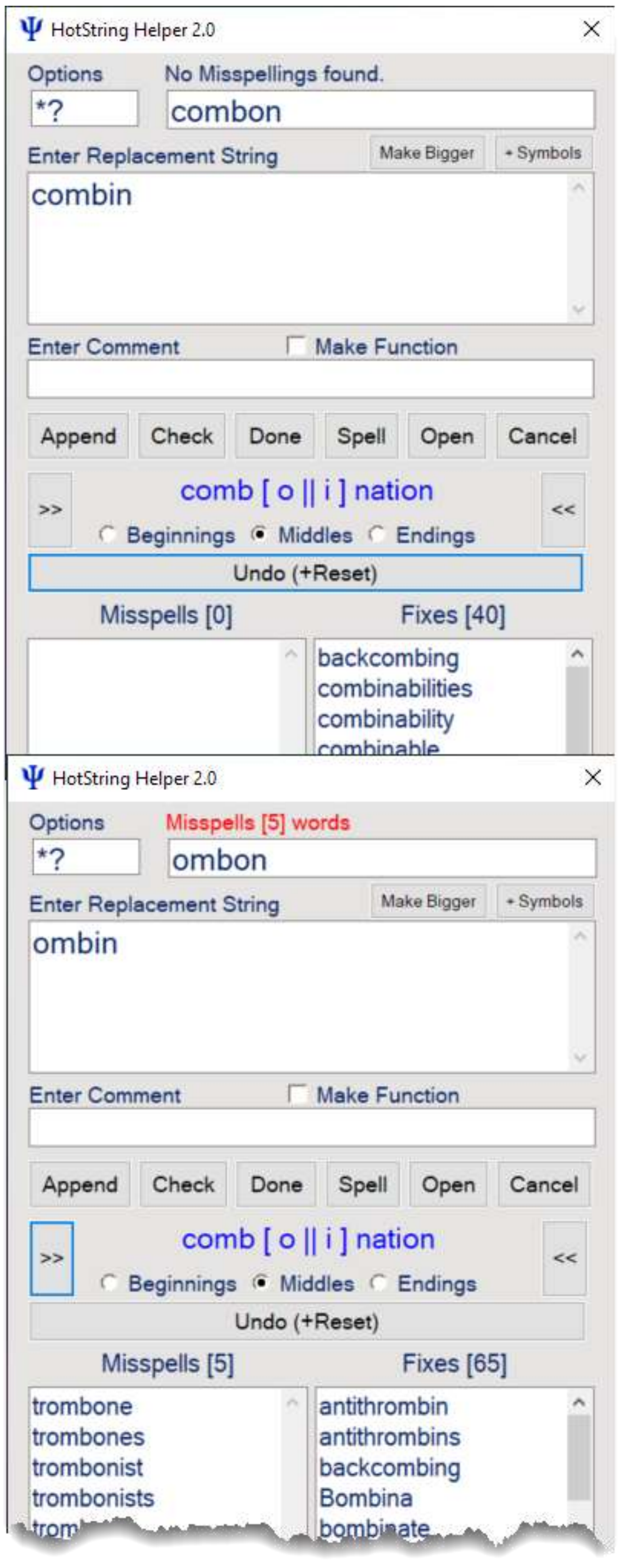
Characters that are -common to the beginning of both strings [-specific to trigger | | -specific to replacement] -common to the ending of both. Also look at this as **[the error | | the fix]** and at the ends are the parts that are neither error, nor fix.

The bottom-most part of the form has the two lists of word matches. The words that correspond to the trigger string are on the left, and the ones for the replacement string are on the right. The goal is to have *many* words represented on the right, and *none* on the left. The example in the screenshot will potentially fix three different words and won’t misspell any. If the “Misspells” box contains any words, then the Trigger String label will turn red. And will show how many words are matched. This allows the user to be alerted to misspellings, even when the Exam Pane is closed.

It is noteworthy that the Exam Pane is not useful for creating boilerplate template entries. Also, the Make Bigger feature is not useful for autocorrect entries. As such, there is no reason to have both active at the same time. Indeed, *making the hh2 form bigger will hide the Exam Pane, and vice versa.*

With the screenshot on the right, the top part shows the effect of trimming five letters off of the right side and having the *Middles* radio button selected. There are 40 potential word fixes and 0 misspellings.

The bottom part of the image shows the effect of *then* trimming one character from the left. The resulting trigger string ("ombon") is present in the word "trombone" and several other related words. So, it was trimmed too much. I clicked the Undo button, then Appended it to my personal AutoCorrect.ahk file with the Append button. Cool.



Options: *?
No Misspellings found.
Trigger: combon
Replacement: combin
comb [o || i] nation
Beginnings [✓ Middles] Endings
Misspells [0]
Fixes [40]
backcombing
combinabilities
combinability
combinable

Options: *?
Misspells [5] words
Trigger: ombon
Replacement: ombin
comb [o || i] nation
Beginnings [✓ Middles] Endings
Misspells [5]
trombone
trombones
trombonist
trombonists
trom
Fixes [65]
antithrombin
antithrombins
backcombing
Bombina
bombinate

Add-a-Letter

Sometimes you need to add a letter back to the beginning or end of the trigger and replacement. An example happened when I tried to type "Lemmings" and it got erroneously changed to "Lemings." To remedy, I selected the word, pressed Win+H, and ran a Validity Check to get the line number of the bad entry. It was this item:

```
:*?:emmin::emin
```

I like to use the word "Lemming" sometimes, so I had to fix this. Upon expanding the Exam Pane, I saw that **emin** fixes 299 items but **emmin** corresponds to 11 other words. Upon checking the list of 299, I saw that most of those started with the letters *b, d, f, r,* and *s.* And… None of the 11 possible misspell words had those letters for the immediate-left of "emin." So… I systematically and experimentally added each of those letters to the beginning of the string, to see if the resulting autocorrect items were good. To make this process of adding a letter easier, the Replacement string "watches" the Trigger string. If a single letter is added to the beginning/left, or the ending/right of the trigger, then the same character is added to the replacement, in the correct location. An important point, is that this behavior won't be desired when creating Boilerplate Template entries. As such, *the Add-a-Letter feature is only available when the Exam Pane is open.* The process of updating the replacement is slow, so add the letters slowly, or it won't keep up.

Important points:

- They *must* be added to the *very* left or right.
- Add the letters slowly.
- Only use this feature when the Exam Pane is showing.

__**Tip:**__ There is a shortcut key… Pressing **Shift+Left** focuses the Trigger string box and puts the cursor in the Home position.

Check For, and Capture, Existing HotString.

There are two primary ways that a typo-correction pair will get placed into the hh2 form. The above "combonation" example involved selecting the typo with the mouse, then pressing Win+H. It takes time for you to Examine/Analyze a word though. If you are in a hurry, just use the Spell function to correct the spelling, then save the item as a whole-word autocorrect entry. *Then later…* When you have time, launch hh2 and press Open. This takes you to the bottom of your autocorrect list, where the new items are. *Select the entire hotstring (including colons) and press the hotkey.* There is a "*check* *for existing hotstring*" mechanism. If a hotstring found in the selected text, the trigger, replacement, options, and comment will be captured, and get populated back into the hh2 form for analysis. This is done by parsing the existing hotstring with a *Regular Expression*. This particular regex was created by Andymbody and is optimized for efficiency, utility, and ease of use.

The Comparison Word List.

As mentioned above, I'm a big fan of the WordWeb Dictionary PC application. While creating this manual, it occurred to me that I should check with the WordWeb developer about the legality of me sharing hh2 with a word list that was generated from his (commercial) software. Indeed, I learned that the list is "non reproducible," so I've obtained several other similar, *open source*, word lists that will be included in place of the list that was used with the above screenshots.

There is a small text label at the bottom of the Exam Pane, that indicates what the currently assigned word list is. (See screenshot two pages back.)

The to-be-used list of comparison words gets assigned via the "WordListFile" variable, near the top of the code, where the other user options are. Please include the text file extension in the name—such as: **WordListFile := 'GitHubComboList249k.txt'**

If you *double-click the file name* at the bottom of the Exam Pane, then hh2 will attempt to open your AutoCorrect.ahk file in your default editor, then activate "Find" and type the file name in there. Then, it will open the folder of word lists in your file browser. This is intended to make it easier to change the word list if you want to do so.

The Six Buttons.

The six prominent buttons along the bottom of the form are as follows:

Append: Check for validity, then add the current item to the script and reload the script. (Enter key does this too). If Shift is held, item is sent to Clipboard rather than script.

Check: Do a Validity Check, but don't append the item. Validity Checks are discussed in the next section.

Exam/Done: Toggles open/closed the Exam Pane for word analysis.

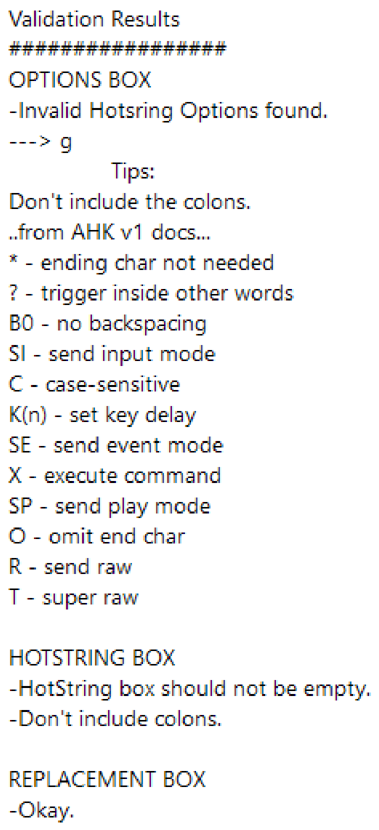
Spell: Attempts to return a Google Search *"Did You mean…?"* result for the Replacement string.

Open: This is for reviewing your recently-appended hotstrings. The Open button will open your AutoCorrect.ahk file in your default ahk editor and browse to the bottom of the file.

Cancel: The Cancel button simply hides the hh2 form and restores the previous clipboard contents.

Validity Checks.

The hh2 tool does several *Validity Checks*. Validity checks occur under two circumstances: (1) If the user presses the Check button, a check is done and a message box such as the one on the right, will provide feedback as to whether there are concerns, or if the Options, HotString, and Replacement Boxes are all "OK." (2) If the user uses the Append button, a check is also done, but a message is only displayed when there are validity concerns. If there *are* concerns, there is an option to "Append anyway."



```
Validation Results
#################
OPTIONS BOX
-Invalid Hotsring Options found.
---> g
            Tips:
Don't include the colons.
..from AHK v1 docs...
* - ending char not needed
? - trigger inside other words
B0 - no backspacing
SI - send input mode
C - case-sensitive
K(n) - set key delay
SE - send event mode
X - execute command
SP - send play mode
O - omit end char
R - send raw
T - super raw

HOTSTRING BOX
-HotString box should not be empty.
-Don't include colons.

REPLACEMENT BOX
-Okay.
```

The validation checks, in order of occurrence, are as follows:

OPTIONS BOX.

Valid AutoHotkey hotstring options include such things as

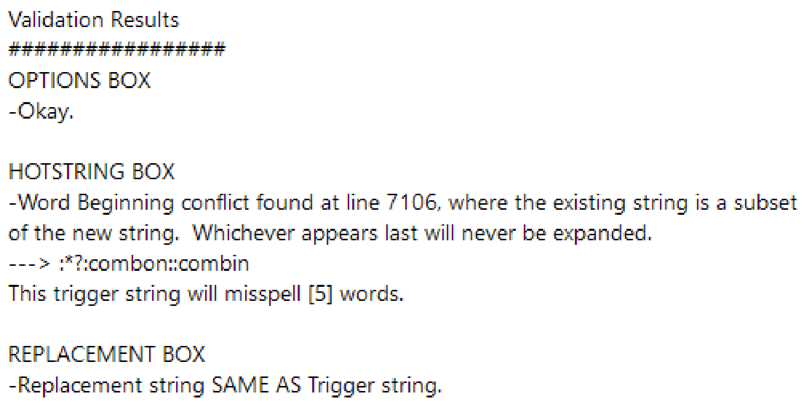
* ? B0 X T C

and appear between the first double colons. The colons, themselves, are not options and should not be included in the Options box. The code will flag any characters that are not valid options (such as "g" in the top screenshot to the right). It will also provide some brief *Tips* that are based on the AutoHotkey Documentation.

HOTSTRING BOX.

Most of the validation mechanism is dedicated to checking the trigger string, i.e. the "hotstring."

*If the trigger string edit box is blank, it will warn you.
*If you are about to create an entry with a trigger that is a *Duplicate* of an existing trigger, it will warn you.
*If the new trigger string is one of the four possible subset/superset conflict scenarios (such as the bottom screenshot on the right), it will warn you. There is more discussion about these conflict scenarios below. Currently (Jan 2024) hh2 does not identify the "*special situations*" for word endings discussed below.
*If the new trigger string is likely to inadvertently misspell other words (also shown in screenshot), it will warn you.



```
Validation Results
#################
OPTIONS BOX
-Okay.

HOTSTRING BOX
-Word Beginning conflict found at line 7106, where the existing string is a subset
of the new string.  Whichever appears last will never be expanded.
---> :*?:combon::combin
This trigger string will misspell [5] words.

REPLACEMENT BOX
-Replacement string SAME AS Trigger string.
```

REPLACEMENT BOX.

With the replacement box, the following will result in a warning:

*The string starts with a colon.
*The box is blank.
*The string is identical to the trigger string (as in the screenshot).

Conflict Scenarios.

Our previous philosophical discussion of what makes a good hotstring might have also included*, "Is not a duplicate of, and does not conflict with, an existing hotstring."* Consider the following pairs of (fake/sample) autocorrect entries.

```
; Word Beginning conflict (nullification?):
; wizard2 never gets used, because typing "wizer" activates wizar1.
; The order of these do not matter.
:*:wizer::wizar1
::wizerd::wizard2 ; Is nullified.

; Word Middle conflict (nullification?):
; Just like word Beginning conflict...
; experiment1 never gets used, because typing "expire" activates xperi2.
; The order of these do not matter.
::expirement::experiment1 ; Is nullified.
:*?:xpire::xperi2

; Word Ending conflict:
; Typing "likour " will result in one, or the other, WHICHEVER is first.
; The second/bottom one is never used.  Order matters here.
::likour::liquor1
:?:kour::quor2
```

First let's point out that there might be legitimate reasons for duplicate replacement strings. For example, the two next items appear in the original 2007 AutoCorrect.ahk.

```
::teh::the
::hte::the
```

These are both common ways to misspell "the." They are not duplicate hotstrings, because the triggers are different. Now consider these two:

```
::theer::there
::theer::three
```

These fix different words, so the autocorrect items, *in whole*, are not duplicates, but the duplicate *hotstrings* conflict with each other. Whichever appears first in the script file will be loaded into RAM by AutoHotkey (v1 or v2) and made available for use. The second one will be ignored. Regarding the fake items at the top of the page: With the first pair, there is a word-beginning item (opts: *), whose trigger is a *substring* of the no-options one near it. Importantly, the substring is a "left-match" of the full string, i.e., the superset has additional characters *at the end* of the trigger string. These characters at the end will never be seen by AutoHotkey, because typing the first part of the trigger activates the "substring-match" item with the asterisk in the options (remember, * = no End Char is needed to activate the hotstring). The next pair has a word-middle (opts: *?), which works similarly. With either of these, the order doesn't matter. The shorter one will always supersede the longer one *unless* there is a *Context Sensitive* section in your script. We'll briefly look at Context Sensitive hotstrings below, but to conclude this section, just remember that word-beginnings and middles don't play nice with other hotstrings that are supersets. I'm not even sure that this should be called a "Conflict." It's really more of a "*Nullification*." Word-endings on the other hand,

```
:?:toin::tion
:?:llly::lly
```

actually do conflict with each other, in the sense that "*Whoever is first, wins."* The order does matter. It is also relevant that, under certain circumstances when working with word-ending hotstrings, (1) conflicting items can both be used, and (2) potential misspellings can be nullified.

Special situations with word-ending entries.

The previous section introduced how word-ending-match autocorrect items can conflict with each other in the sense that the second one is never seen. There are exceptions. Consider the below pair of items from my current AutoCorrect list.

```
:?*:ngiht::night ; Fixes 103 words
:?*:iht::ith ; Fixes 560 words
```

In my own list they are embedded in f() functions (discussed below), and they are not right next to each other. They *are* in the same order though! The number of potential fixes reported, will depend on the word list used. Note also that as word-endings, there would be much fewer potential fixes. The 103 and 560 totals assume that these items are word-middles. *The relevant point here*, is that ***when*** there are "overlapping" word-ending items, where one is a subset, and is a right-most-match, ***if*** the longer one is first, then they both will be active. You can paste these into an .ahk file and experiment if you want (exit AutoCorrect.ahk first, if it's running). Now (mis)type "goodngiht" and you'll get "goodnight." (mis)Type "wiht" and you get "with." However, if the shorter substring item is first, then the longer one doesn't work. In that case, (mis)typing "goodngiht" yields "goodngith". The hh2 Validity check mechanism is not detailed enough to recognize these intricacies. It will just report, "Word-ending conflict."

The other special situation works similarly. In this case, the longer word is one that we want to preserve and prevent it from getting misspelled. The items below are adapted from Jim B's original 2007 AutoCorrect.ahk.

```
:B0:design:: ; The B0 option turns off backspacing,
:B0:feign::  ; so the typed word just stays there.
:B0:resign::
:B0:sign::
:B0:sovereign::
{ ; <--- braces needed for v2, but not v1 AHK.
        return  ; This makes the above hotstrings do nothing
}                ; so that they override the ign->ing rule below.
:?:ign::ing
```

Notice the word-ending fix for "*ing*" at the bottom. It's interesting that there have been several articles and blog posts about the original AutoCorrect.ahk script over the years. It usually gets described as "*The script fixes some 4700 common misspellings and typos*." In reality though… If you count the ign -> ing fix, the number of corrections is much higher! Depending on what word list you compare against, the number of potential corrections might be from 7 to 15 thousand just for that one hotstring. Unfortunately, it also misspells 40 or so words, such as "sign." (Not all are shown above.) With the above chunk of code, you'll notice that, again, the longer-string-items appear first. The "B0" option tells AutoHotkey to not remove the trigger string. And since there's no replacement, the typed word just remains unchanged. Typing "ign" by itself, or following any characters that are not one of the preceding longer strings will trigger the replacement. The longer, preceding strings are "protective" in the sense that they protect us from "miscorrecting" the words. A relevant topic that is related to this is the topic of "sacrificial words."

When misspellings are tolerated.

As discussed above, a good autocorrect entry corrects lots of common words, but *doesn't misspell other words* in the process. But what happens when a potential entry corrects many common words, but misspells one or two ***uncommon*** words? Do we sacrifice that word; knowing that if we ever type it, we'll have to press Ctrl+Z to undo the (mis)correction? Consider this example:

```
:?*:ahve::have ; Fixes 47 words, but misspells Ahvenanmaa, Jahvey, Wahvey, Yahve, Yahveh (All are different Hebrew names for God.)
```

The entry "::ahve::have" is one of the whole-word AutoCorrect 2007 items. By simply adding the "?*" it matches 47 English words. As it turns out, there isn't a single English word with "ahv" in it. Unfortunately, that three-letter combination does appear in several of the Hebrew names for God. If a person is a Jewish scholar (that writes in English), they will definitely be interested in this dilemma.

Here is another example:

```
:?:itr::it  ; Fixes 366 words but misspells Savitr (Important Hindu god)
```

By using the same technique that Jim did with ":?:ign::ing" we can "protect" some of these words. Unfortunately, it only works with word-endings. It does not work for word-middles or -beginnings (i.e. Ones that have an asterisk in the options.) So, this uncommon word:

```
:B0:Savitr::
```

gets protected. However the words,

```
:B0:Ahvenanmaa::
:B0:Jahvey::
:B0:Wahvey::
:B0:Yahve::
:B0:Yahveh::
```

don't get protected because the corresponding hotstring ":?*:ahve::have" doesn't allow it. I put all of these near the top of the autocorrect list, then commented-out the ones that won't work. There are 37 "good" protective strings and 35 commented-out "no good" ones. Fortunately, they really are pretty obscure words. But, as indicated in the code comments: "***If you hope to ever type any of these words, locate the corresponding autocorrect item and delete it***." Both sub lists are sorted alphabetically. For the above examples, I took the first item from each. It was just a coincidence that both "sacrificial" words were religious words. But… Since we're doing religious words…

Using Case Sensitivity to protect words.

Consider this example:

```
:*C:carmel::caramel ; Fixes 12 words.  Case sensitive to not misspell Carmelite (Roman Catholic friar)
```

This was made from the 2007 whole-word entry, "::carmalite::Carmelite". Believe it or not, there are 12 English words that have caramel as a root at the beginning. Adding the asterisk allows the hotstring to match any of them. Unfortunately, a Roman Catholic friar is (apparently) called a "Carmelite." I looked it up in the dictionary, and it appears to always be capitalized. So, we add the "C" option. Now if we type the word with a lower-case "c," which matches the trigger, it gets changed. If we use upper case, it does not get changed. Unfortunately, this also means that **if** I write a sentence about caramel, **and** I start the sentence with the word caramel, **and** I misspell it as the trigger string, it won't get corrected. I guess that's another "sacrifice."

For most purposes, this *case-sensitivity-to-protect-words* technique only works with word beginnings. When the to-be-protected word is an acronym (or initialism), then word-endings or -middles might also apply. Here is an example:

```
:?C:hc::ch ; Fixes 446 words, :C: so not to break THC or LHC
```

This one is "safer" in the sense that a misspelled word at the beginning of a sentence will still get corrected; Unless, of course, we are writing in all caps. An opposite example might be:

```
:C:ASS::ADD ; Case-sensitive to fix acronym, but not word.
```

As a school psychologist who often writes reports for kids with Attention Deficit Disorder, this autocorrect can prevent a very embarrassing typo. On the other hand, if I were an inflamed internet troll, I may want to remove this item from my list.

Context Sensitive Hotstrings.

Two pages back, when discussing the problem of duplicate hotstrings, we mentioned Context Sensitivity. AutoHotkey does allow us to create *Context Sensitive* hotstrings and hotkeys. A self-explanatory example right out of the documentation is as follows:

```
#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.
```

The hh2 app does not create Context Sensitive hotstrings. Also, it is important to note that the above-discussed validity checking will not recognize if a hotstring is embedded between #HotIf directives. It will report, "Duplicate found," even though duplicates are okay, when one is restricted to certain contexts (which are usually particular windows). There is, however, a workaround… Just before the hotstring definitions, there is a function called “getStartLineNumber().” Validity checks only happen below this point, so have the #HotIf items above it.

Inevitable Errors.

Despite our best efforts to create hotstrings that don’t have triggers corresponding to other words, there will inevitably be times when errors will occur. In my own experience, I believe there are two scenarios in which this occurred the most. <u>Keyboard input buffering problems</u> and <u>confounding mis-typings</u>.

Regarding keyboard input buffering

Consider this test hotstring:

```
:?*:zx::lllllllllllllllllllllllllllllllllllllllllllllllll
```

Put this in a script and try it. You’ll notice that “zxcv” are next to each other on the keyboard, so you can type them by “strumming” the fingers of your left hand. Try it several times… This is what I get:

```
lllllllllllllllllllllllllllllllllllllllclllllllllllv
lllllllllllllllllllllllllllllllllllllllllllcllllllv
lllllllllllllllllllllllllllllllllllllllllcllllllllv
llllllllllllllllllllllllllllv
lllllllllllclllllllllllllllllllllllllllllllv
lllllllllllllllllllllllllllllllllllllllllcllllllllv
lllllllllllllllllllllllllllllllllllllllllllllllcllv
```

The AutoHotkey application is written such that hotstring execution is intended to be buffered. This means that, once the hotstring is activated, any additional keypresses should be held in RAM, then sent *after* the hotstring is finished. The resulting expanded text *should* look like this:

```
lllllllllllllllllllllllllllllllllllllllllllllllllcv
lllllllllllllllllllllllllllllllllllllllllllllllllcv
lllllllllllllllllllllllllllllllllllllllllllllllllcv
lllllllllllllllllllllllllllllllllllllllllllllllllcv
```

Communication with the current AHK developer (*Steve Gray aka Lexikos*), suggests that other applications can compete for the keyboard buffer and cause the interspersed effect. This example has a really long replacement string, but it can happen with shorter strings as well.

I find that, when expanding a boilerplate entry, I tend to sit there and wait for it to play back before I continue typing. *With autocorrects, however,* a person doesn’t know when they will occur, so there isn’t a way to pause for them… You just keep right on typing, which invites buffering glitches.

Regarding confounding mis-typings

Consider this autocorrect word-middle item:

```
:?*:cirp::crip ; Fixes 126 words, but misspells Scirpus (Rhizomatous perennial grasslike herbs)
```

Now imagine I intend to type “chirp,” but I mistype it, “cirp.” Since that error string happens to be a trigger for a different word, my AutoCorrect tool erroneously changes it to crip. *“I heard the robin crip.”* I think this word-middle autocorrect entry was probably generated from the 2007 whole word item, “::scirpt::script.” There’s a trade-off here, shorter trigger strings (as word parts) match more words. But… longer trigger strings are less likely to be accidentally typed during a confounding misspelling. Which one should be used?

-longer strings = unambiguous
-shorter strings = more potential matches

An argument can be made for each, but for the most part, I leaned more toward shorter strings with more matches, when processing the 2007 AutoCorrect and the Wikipedia grammar items. Still, it bothers me when my AutoCorrect script makes changes and I can't keep track of, or even detect, them. I needed a way to log the corrections.

AutoCorrection Logging.

I don't know who the first person was to use *Function Calls* for individual AutoCorrect items, but I got the idea from Mikeyww. The dilemma was how to capture the most recently used trigger into a variable so that it could be viewed. The AutoHotkey application has several built-in variables (**A_Vars**), that capture various metadata about running scripts. In fact, there is an **A_ThisHotkey** variable that contains the most-recently-used hotkey. It will also work for hotstrings, but… not for *auto-replace* hotstrings, which is exactly what is needed for this purpose. So how do we collect that information? The solution is a function-calling hotstring. It won't work if you have this:

```
::tpyo::f("typo")
```

because this will only replace "tpyo" with the text "*f("typo")."* However, adding an "X" in the options tells AutoHotkey to treat the "replacement" section as computer code. It's "X for eXecute."

This version:

```
:X:tpyo::f("typo")
```

Will indeed call the function (assuming there is a function f() defined somewhere). Remember that we wanted to capture that last used hotstring trigger. By adding the X option, this is no longer an *autoreplace* hotstring, so we can send the variable content to the function to be used.

```
:X:tpyo::f("typo", A_ThisHotkey)
:X:hello wrold::f("hello world", A_ThisHotkey)
f(replacement, trigger)
{   Send replacement
    Last_used_trigger := trigger
}
+!F3:: MsgBox Last_used_trigger
```

Now press **Shift+Alt+F3** to see the last used hotstring trigger. Cool. It occurred to me later that I needed at third parameter, to hold the *End Character*, so that it could be Sent, following the replacement text. Rather than simply hold (only) the last-used trigger string, it also occurred to me to append them to a Log File for later analysis.

The AutoCorrectsLog File.

When an auto replacement is made, the function beeps, then waits for one second, and logs the item by appending it to the bottom of the *AutoCorrectsLog.ahk* file. One line per item, formatted with the date as MM-DD. The reason for waiting one second is to see if the user presses the Backspace key. This information is recorded in the log file. If there is a normal double-hyphen "--" separating the date and string, then Backspace was not pressed during the one-second timeout. If there is a left arrow "<<" then Backspace *was* pressed.

Example:

```
01-19 << :?*:voiu::viou
01-19 -- :?*:cuas::caus
```

The assumption here, is that, if an autocorrect item "goes rogue" then you'll immediately press Backspace to correct it. Unfortunately, this is not a perfect system. For example, when I'm tired and busy and distracted, my typing can get pretty sloppy. It is entirely possible that the autocorrect will make the desired correction, but I'll also make an *additional* error—and backspace it—all within one second. Also, it's noteworthy that I tend to watch the computer screen as I type. This is true for most people, but not all. I suspect that most people who, for whatever reason, don't monitor their typing, probably won't want to use an autocorrect app.

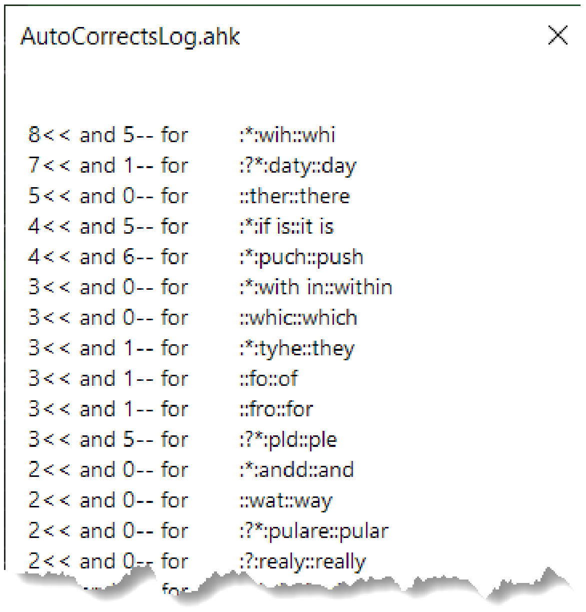
After logging quite a few autocorrections, I wanted to generate frequency reports to analyze the data. I decided to go ahead and embed the code for that right in the log file. You'll notice that the above log examples are not "AHK computer code" so I put a start-block-comment declaration:

```
/*
```

AutoHotkey considers everything below that (i.e. the log) to be comments.

Log Analysis.

Pressing **Shift+Alt+L** will run the log file. If you have a normal installation of AutoHotkey, then this will probably run the analysis. Otherwise, it might open the log file in your default editor.



AutoCorrectsLog.ahk

```
8<< and 5-- for      :*:wih::whi
7<< and 1-- for      :?*:daty::day
5<< and 0-- for      ::ther::there
4<< and 5-- for      :*:if is::it is
4<< and 6-- for      :*:puch::push
3<< and 0-- for      :*:with in::within
3<< and 0-- for      ::whic::which
3<< and 1-- for      :*:tyhe::they
3<< and 1-- for      ::fo::of
3<< and 1-- for      ::fro::for
3<< and 5-- for      :?*:pld::ple
2<< and 0-- for      :*:andd::and
2<< and 0-- for      ::wat::way
2<< and 0-- for      :?*:pulare::pular
2<< and 0-- for      :?:realy::really
```

The image to the right shows what the reported information looks like. The items are sorted by how many times they were Backspaced. Only the top 40 items are reported. The number of times the same item was *not* Backspaced is provided for comparison. The item,

```
8<< and 5—for :*:wih::whi
```

means that this autocorrection was logged 13 times. Eight of those times, a Backspace occurred within one second. The other five times, it did not.

Check out the "*daty -> day*" item. It was removed seven times, and only kept once. Presumably, it is getting triggered when it's not supposed to. That's unfortunate, given that it potentially fixes 168 different words.

I just now changed it in my AutoCorrect list, so that it is a "word end" rather than a "word middle." We'll see if that helps. *I'll leave my logged items in there for you to experiment, but obviously, you'll want to delete them and collect your own data.* Running the analysis does take time… On my laptop, 1300 items takes about one-and-a-half seconds. I did a text run with 10k items though, and that took 44 seconds.

The logging process is useful, but I have to admit that it is not the thing which has had the biggest impact on increasing the reliability of my system. I previously had two different logs collecting different levels of contextual information. Even will all that collected data, I was still seeing enough errors that I had discontinued using a previous version of this script. Then, Descolada made his *InputBuffer Class* and shared it. This made a substantial difference, and I saw, perhaps, 80 or 90 percent fewer errors! After that, I readopted the current script and have been happy with it. The InputBuffer Class is now embedded in AutoCorrect for v2, and it gets used by the f() function. I did remove the *Mouse Buffering* elements from it, since the f() function doesn't do anything with the mouse.

HotString Rarefication.

While working with some of the longer "Grammar corrections" garnered from Wikipedia's *Lists_of_common_misspellings/Grammar_and_miscellaneous*, it occurred to me that I could "**Rarify**" some of them to make them leaner… More streamlined. Consider this hotstring:

```
::agreement in principal::agreement in principle
```

When auto-replacing a hotstring, the normal operation is for AutoHotkey to "Backspace away" the trigger string, then type the replacement. So, the above hotstring, as it is written, will instruct the AutoHotkey app to send Backspace 22 times, then type out the replacement. This is largely superfluous, because only the last two characters are actually different between the trigger and the replacement. So why not just press Backspace twice, then type the last two characters of the replacement string? We could turn off automatic backspacing with the "B0" (b zero) hotstring option and setup the autocorrect item like this:

```
:B0:agreement in principal::{BS 2}le
```

However, without the asterisk, an End Char is needed to trigger the string, so we need to "remember" what End Char was used and put it back. For that, we need to use the X option. Either of these will work:

```
:B0*:agreement in principal::{BS 2}le
:B0X:agreement in principal::SendInput "{BS 2}le" A_EndChar
```

The f() function, as it currently exists, will automatically, at runtime, compare the trigger and replacement strings and will only remove the necessary characters. The hotstrings don't have to be set up as in the above examples. This just happens automatically every time the function is called.

The f() Function.

The function as it currently exists, takes three parameters. Only the first one (the replacement/expansion string) varies from item to item.

```
:B0X:trigger::f("replacement", A_ThisHotkey, A_EndChar)
```

What the function does:
- Call the InputBuffer, which captures any manual keypresses.
- Save the hotstring to a variable so we can peek at it and/or log it.
- Rarify the replacement and determine the number of backspaced needed.
- Type the necessary backspaces and needed part of the replacement string.
- End the Buffer, thus typing any captured manual keypresses.
- Announce the replacement with a beep.
- Log the item, (via calling the GoLogger function) indicating whether Backspace was manually pressed by the user within one second.

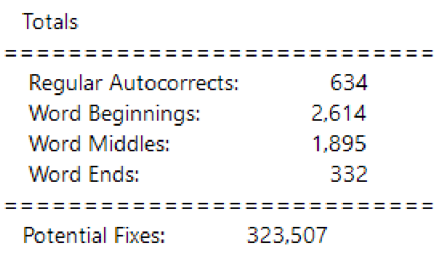
As seen in the above screenshots, there is a **[ ] Make Function** checkbox under the Replacement String box, but above the Comment box. Check the box if you'd like hh2 to format the new AutoCorrect entry with the f() function syntax. If the box is unchecked, a normal hotstring will be created. Should the checkbox be checked by default? In the part of the code where the hh GUI is made, is the following line of code:

```
ChkFunc.Value := 1
```

That's a 'number one.' Change it to a 'zero,' or delete that line of code to cause the *Make Function checkbox* to be *unchecked* at startup.

AutoCorrect-Fixes Report

In the code, before the list of AutoCorrect items, but after the InputBuffer Class, is a small utility to total a few things. Pressing **Ctrl+F3** provides the mini-report on the right. "Regular Autocorrects" refers to whole-word items. Beginning, Middle, and End items are the titular multi-match items. When I started manually converting the 2007 AutoCorrect list, to see how many multi-word matches could be made, I flagged some of the higher-utility items with "Fixes X words" as an inline comment. Upon creating the previously-mentioned Word Analysis GUI Tool, I automated the process of adding the flag. Once practically every item was appended with its corresponding number of potential fixes, it made sense to automate the process of tallying those. Mikeyww helped with the original scriptlet. As seen in the screenshot, the current version of AutoCorrect here can potentially fix >323k items, which is pretty good. Note that the digit column appears right-justified, but I "faked" that effect by putting extra spaces in front of the 3-digit numbers.



```
Totals
============================
 Regular Autocorrects:        634
 Word Beginnings:           2,614
 Word Middles:              1,895
 Word Ends:                   332
============================
Potential Fixes:          323,507
```

Some Notes and Caveats…

* The "323k" claim doesn't mean that 323k unique words can be fixed. Indeed, the individual items were converted using the word list from the WordWeb app. The list had about 186k words… So how can there be 323k possible fixes? It is because there are

several different *and common* ways to misspell any given word. Also, one word-middle item might fix part of a word, and another might fix a different part of the same word. So, there is a great deal of overlap between the words, in terms of which words can potentially be corrected.

* Another important point is that the total number of potential matches—and—the number of possible misspellings, will vary *greatly* depending on which word list is used as a comparison. Obviously, the word list with 466k words will indicate more matches than the one with 26k words.

* It's worth noting that I frequently reference Jim's 2007 AutoCorrect.ahk list, which is based on the Wikipedia list, and the grammar items, based on another Wikipedia list. Indeed, perhaps 95% of the items came from those two sources. In reality though, the process of winnowing-down the words to create multi-match items loses the reference to the original word. So, my version bears little resemblance to the original lists, and it would be impossible to reverse the conversion process, to get the original items back. As you might guess, *many* of the items in AutoCorrect 2007 were also removed, because there were redundant root words with different prefixes and suffixes. It was usually the root word (or part of it) that got used for the multi-match items, so the redundant whole-word items got culled.

* It's also noteworthy that most of the grammar-fix items are comprised of multiple words. Since none of my word lists (which I used to calculate potential word matches/fixes) include multi-word phrases, there's really no easy way to programmatically determine the number of potential fixes per item. As such, most of these items are just flagged with "Fixes 1 word."

* It is relevant that the original AutoCorrect 2007 has 262 commented-out "ambiguous" items, such as, "::wich::which, witch." What should the replacement be? "which" or "witch?" I chose "which" because it is a word I use more often. I also picked my preferred replacement for most of the other ambiguous items as well. Many of the ambiguous entries were ambiguous because of British vs. American English. My apologies go to British users who keep getting "Americanized" autocorrections.

* The list is not perfect! In the "Rarification" section, I use the example of

```
::agreement in principal::agreement in principle
```

Upon proofreading this manual, it occurred to me that

```
::in principal::in principle
```

would be a higher-utility item because it matches "agree in principal," and many other phrases. I searched the list, however, and found that both items, in addition to

```
::agree in principal::agree in principle
```

were already present. The "in principle" one makes the other two superfluous.

* It's hard to search your AutoCorrect.ahk file for the typos! If I open my script, then press Ctrl+F to "find" something, then attempt to type one of the trigger strings (assuming my script is running), it will be corrected. To avoid this, type half of the trigger, then press Left Arrow, then Right Arrow, then type the rest of the trigger string. This "breaks" the sequence. Other things will to this too, such as mouse clicks or pressing Esc.

In The Script.

Some other things present in the script are as follows:

The CAse COrrector tool.

The original 2007 AutoCorrect.ahk script included a tool to *AUto-COrrect TWo COnsecutive CApitals* that was originally created by Laszo. It was a clever idea, but had problems, and so was commented-out by default in AutoCorrect 2007. As presented in the AutoCorrect for v2 thread:

I can't be the only one who has erroneously typed "THanks" more times than I've typed the intended "Thanks." It's a useful tool! The original was made by Laszlo, using a loop to monitor for hotkeys. It was a smart idea, but would misfire under certain conditions, and so is commented-out, by default, in the original AutoCorrect.ahk script. A version posted and discussed here was made by myself (with much help) using the InputHook() function. This was much better (I've been using it for months) but it introduced a couple of *other limitations (as seen in the forum thread). Just a couple of days ago, forum member Ntepa posted an even better version (same forum thread). His uses a hotstring loop combined with InputHook and addresses the limitations of my own version, as well as adding a couple of additional improvements. It was so superior to mine that I removed my own version and included his (with permission) in the above code. Tip: Ntepa has cleverly included a 400 ms timeout because the double-capital effect usually occurs from typing too fast. If a person wants to experiment with the tool, they might like to temporarily remove the timeout. Do this by removing the T.4 from InputHook("V I101 L1 T.4"). You could also shorten/lengthen the timeout to fit your typing/typo speed.*

Unfortunately, none of the above versions are completely error free. Even with the latest version, I occasionally have a mid-sentence proper noun erroneously converted to lowercase. I'm still working on figuring out why this happens. To help figure this out, I've been including the Active Window when logging CapFixes. Please note that the above-mentioned "Log Analysis" does not analyze automatic Cap Fixes. Those have to be reviewed by looking in the actual log items.

I previously had the *CAse COrrector* tool and *AutoCorrect* is separate script files for a while, but I've recently combined them so that they could share the log function "GoLogger()."

It's noteworthy that this tool will correct several two-letter words such as "IT---> It." As indicated in a recent forum post: There are several common two-letter words in the RegEx part of this line of code:

```
|| (char3 = A_Space && char1 char2 ~= "OF|TO|IN|IT|IS|AS|AT|WE|HE|BY|ON|BE|NO")
```

Ntepa added those for us so that they would get corrected AA --> Aa even though they don't conform to the *Upper Upper Lower* rule. You can get rid of that functionality by removing that part of the code. Be sure to remove the **||** but don't remove the brace **{** at the end of the line.

Accented words with diacritic characters.

The AutoCorrect 2007 script includes 287 *accented words*, such as the example below. As with this example, many of the items have plural forms, so I made them word-beginning items. I don't consider these to be "typos or misspellings" so I didn't embed them in function-calls. I thought it would be fun to tag each item with its dictionary definition.

```
:*:angstrom::Ångström ; noun a metric unit of length equal to one ten billionth of a meter (or 0.0001 micron); used to specify
wavelengths of electromagnetic radiation
```

Most of the definitions were obtained "in bulk" from https://www.easydefine.com/. Others are from the WordWeb application.

Above, on page 6, we discussed the "Show Symbols" button of HotString Helper 2.0. The *diacritic characters* in the accented words are similar to this, in that they require your .ahk file to be saved in "Unicode" format for the characters to be displayed correctly. As indicated in the AutoCorrect for v2 forum thread, I recommend this:

1) Open plain old Notepad.
2) Do "Save As"
3) Change "Save as type" from txt to "All files (.)"
4) Change the Encoding to "UTF-8 with BOM."
5) Save (using .ahk extension) /close/reopen.
6) Paste in the code that contains the accented words and save again.

End

Please post a reply on the AutoCorrect for v2 forum thread if you find any errors or have any suggestions for the above discussed code or for this manual.

Thanks for reading, and a big THANK YOU to the people who have helped me create this.

-kunkel321