VoiceAttack

www.voiceattack.com

Contents

VoiceAttack Quick Start Guide (v1.8.8+) 3
VoiceAttack Screen Guide 5
VoiceAttack's Main Screen 5
Profile and Profile Options Screens 8
Command Screen 16
Key Press Screen 37
Pause Screen 40
Variable Pause Screen 40
Other Stuff Screen 41
Key Press / Mouse Event Recorder Screen 104
Mouse Action Screen 106
Registration Screen 110
Options Screen 111
Exporting Profiles 129
Creating Quick-Reference Lists 131
Importing Profiles and Profile Packages 132
Importing Individual Commands 133
No Speech Engine / Alternate Speech Engines in VoiceAttack 134
Using the Condition Builder 136
Command Line Options 142
Text (and Text-To-Speech) Tokens 148
VoiceAttack Path Tokens 178
Quick Input, Variable Keypress and Hotkey Key Indicators 179
Key State Token Parameter Values 182
VoiceAttack Plugins (for the truly mad) 185
VoiceAttack Load Options Screen 224
Advanced Variable Control (Scope) 226
Application Focus (Process Target) Guide 229
Command Execution Queues Overview 236
VoiceAttack Profile Package Reference 239
Troubleshooting Guide 241
Setting up Microphone Input 244
VoiceAttack’s Data Storage 245
VoiceAttack Author Flags 246
For fun... maybe 250

VoiceAttack Quick Start Guide (v1.8.8+)

A few things that you'll need for VoiceAttack to work:

1) Microsoft Windows Vista, 7, 8, 8.1, 10 or XP. Windows Vista and up come with the Windows Speech Recognition Engine built in. Windows XP, by default, does not. If your copy of Windows XP does not have these components, you will need to download them from the VoiceAttack site:

   http://www.voiceattack.com/download_for_xp.aspx

   You will know right away when you launch VoiceAttack if you do not have the Windows Speech Recognition Engine :) Note: a link will appear with the same address listed above.

   If you feel adventurous and find out that VoiceAttack works on other versions of Windows, please let us know & we'll make sure to update this document.

2) The .Net Framework v4.7.2. This is a requirement. The installer will show you where to get it if you don't already have it.

3) A microphone... Although not technically REQUIRED for the program to run, you're not going to get very far without one. A USB headset is recommended, since you are probably going to use VoiceAttack to play games. **Setting up your microphone properly** to work with Windows' speech recognition is a very vital step. There is a short section on how to do this, near the end of this document (See, 'Setting up Microphone Input').

4) A voice (see #3). :)

5) Your Windows Speech Recognition Engine needs to be trained up. Again, not an absolute requirement, however, the difference between a trained and untrained system is like night and day.

   Hint: Start in Control Panel... I ran the trainer three times in a row & now recognition works great!

Getting things going...

Once you've got VoiceAttack installed and you have found out that you meet all the requirements outlined above, you can just jump right in. To keep things simple, we're going to assume that you are working with the trial version of VoiceAttack, and this is your first time running VoiceAttack on this computer. If your microphone is on and the input volume of your microphone is properly set, you should see VoiceAttack's Level bar moving when you speak. If the Level bar is not moving, VoiceAttack can't hear you. See the 'Troubleshooting Guide' at the end of this document.

Notice that VoiceAttack is pretty much not recognizing anything you say. This is good, because we have not added any commands to the profile. Click the 'Edit' button to view the commands for this profile.

What is a command? A command is simply a word or phrase you are going to say to VoiceAttack. When VoiceAttack recognizes the command that you say, it will perform a series of actions (which can be keyboard key presses, pauses, mouse clicks, application launches,

sound effects, etc.).

Notice that VoiceAttack has a set of commands already set up for demonstration purposes. You can edit and/or remove all of these items. Let's try one out! Hit the, 'Cancel' button to go back to the Main screen. Now, you must speak into your microphone. Say the word, 'Calculator'. If everything is lined up right (microphone is on, you have adequate volume and your speech recognition engine is trained up), you should see the Windows calculator on your screen. Now say, 'Close Calculator'. The calculator should now be closed. If you are not seeing the Windows calculator, please refer to the Troubleshooting Guide at the end of this document.

Note that this help file is kind of basic and doesn't explain some things in super deep detail. Swing by the VoiceAttack user forum or Discord server for more detailed explanations/conversations about pretty much everything:

http://voiceattack.com/forum

http://voiceattack.com/discord

Thank you for trying VoiceAttack!

PS - This help document is also available online, so please save a tree by not printing this huge document that changes a lot (pretty please).

http://www.voiceattack.com/help

VoiceAttack Screen Guide

This document will make an attempt to explain the main features of VoiceAttack.

VoiceAttack's Main Screen

This is the main hub for all VoiceAttack activity. There's a lot going on here, so, I've numbered the main areas of the screen and describe each part below.

1 - Audio indicator

This icon indicates the status of your commands (recognized, unrecognized, error), as well as a way to tell if your mic is muted.

2 - Options button

Opens the options screen where you will find various settings for VoiceAttack (See 'Options Screen'). Additionally, the registration screens for VoiceAttack are available through this button (See 'Registration Screen').

3 - Profile management buttons

These two buttons will allow you to edit, delete, export and duplicate your currently-selected profile or create or import a new profile. Note: Creating, importing, deleting and duplicating profiles is only available in the registered version of VoiceAttack.

4 - Profile selector

Drop down the list to select one of your created profiles. Each profile contains a set

of commands that you specify.

5 - Listening button

This is a toggle button that enables/disables VoiceAttack's 'listening'. That is, VoiceAttack will stop performing actions on commands that it recognizes. The only commands VoiceAttack will process are the commands that tell VoiceAttack to start listening again (if you specified one or more... see 'Command Screen' for more info). The hotkey for this button can be configured through the VoiceAttack Options screen

Keyboard shortcuts toggle
This toggle button enables/disables VoiceAttack's keyboard shortcuts.

Mouse shortcuts toggle
This toggle button enables/disables VoiceAttack's mouse button shortcuts.

Joystick button toggle
This toggle button enables/disables VoiceAttack's joystick button detection.

Stop Commands button
This will halt all macros that are in progress. Useful if your macros happen to be long-running. Note this will also stop any playing sounds or text-to-speech, and any keys that are pressed down will be released.

6 - Recognition log

This log shows what VoiceAttack is 'hearing' and the actions that VoiceAttack is invoking. This list is quite useful when determining if you need to speak more clearly or rethink the names of your commands. It's also a lot of fun to say weird phrases and see what the speech recognition engine *thinks* you said. Fun for the whole family (maybe). Right-clicking on this log will allow you to copy and clear the text, as well as allow you to specify some options. For instance, you can indicate if new log entries appear at the top or at the bottom of the log, as well as specify how many entries the log will hold. You can also filter out unrecognized commands or just show the latest log entry only. Enabling the, ‘Consolidate Duplicate Log Entries’ option will cause any subsequent duplicate log entries to be consolidated into a single line with a counter (up to 999 items). When this option is not enabled, you will continue to see all log entries, even if they are duplicates.
Note: Right-clicking (or double-clicking) a 'Recognized' log entry will take you to the 'Edit Command' screen for the selected command in the current profile. If the log entry is 'Unrecognized', you will be taken to the 'Add Command' screen. This is to aid in testing new commands and has probably saved me a few pulled hairs. Your mileage may vary :)

7 - Level Bar

A graphical indicator of the microphone input for VoiceAttack. My guess is that the first thing you will do when you are launching VoiceAttack is to look at this bar... and then say, 'hellooooooooooo' into your microphone to see if all is working. Maybe that's just me. Note that this bar will turn red as an additional indicator when listening is turned

off.

8 - Context Menu

The context menu that is accessible from the icon in the top-left corner contains various options and functions:

Always on Top - VoiceAttack will remain as the top-most application while this option is turned on.

Help Document - Access the VoiceAttack help documentation that is located in the VoiceAttack installation directory.

VoiceAttack User Forum - This will launch your browser pointed to the VoiceAttack user forum. Lots of great help from some really great people in there.

Reset Active Profile - This will refresh the active profile, invoking everything that normally occurs in a profile change (without actually changing the profile). This works the same as the, 'Reset Active Profile' action.

Reset Speech Recognition - This item will reset the speech engine that VoiceAttack is currently using. Although a properly configured and working speech engine generally requires no reset during a VoiceAttack running session, this is in place if you need it.

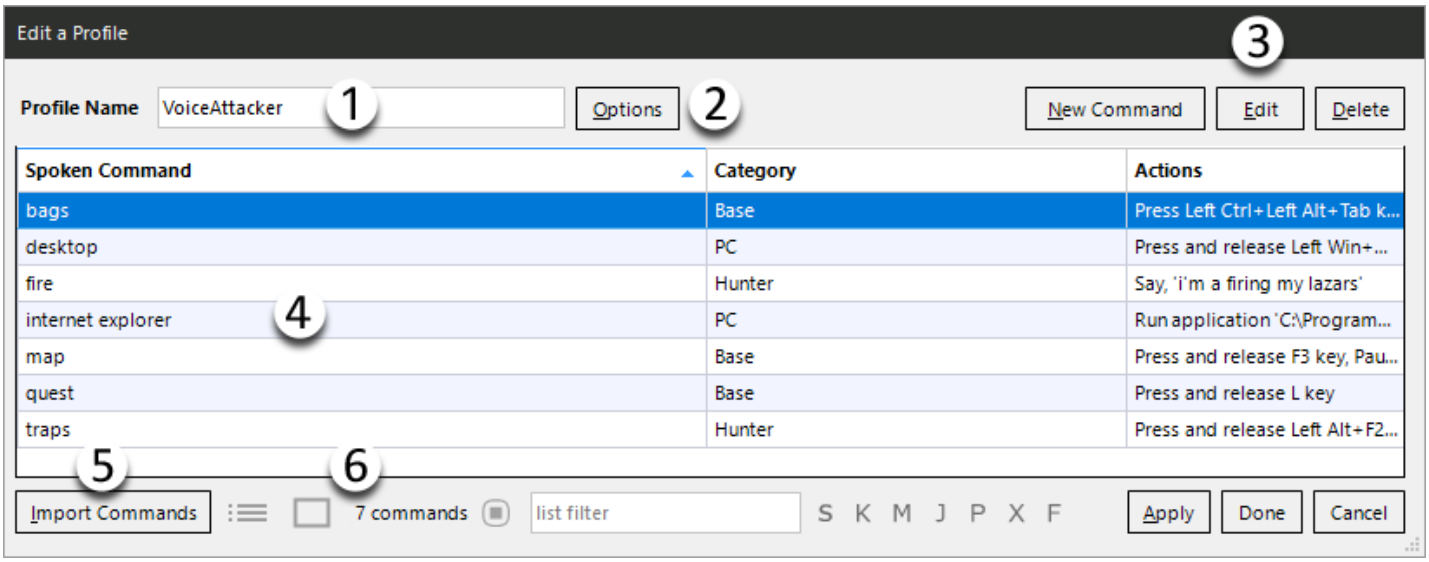
Mute Speech Recognition Device - Checking and unchecking this option mutes and unmutes the recording device that he speech engine is currently using.

Compact Mode – Checking this option will shrink VoiceAttack's main screen down a bit. Unchecking will restore VoiceAttack's main screen to full size.

Cover of Darkness (Dark Mode) – Checking and unchecking this option will toggle VoiceAttack's 'dark' mode for nighttime use.

Profile and Profile Options Screens

This is where you will update each of your profiles. A profile is basically a set of commands that you define.



1 - Profile Name

Type a unique name here for your profile. Make something descriptive!

2 - Profile Options

This is where you can set up some additional attributes of your profile, as well as override some of the global items found in the Options screen (such as the various ways of turning on and off VoiceAttack's listening).

Click this button to go to the profile options/global override screen shown below (four tabs, General, Exec, Hotkeys and Advanced):

Profile Options General Tab

The, '**Override listening if my spoken command begins with...**' option was added as a fun way to interact with VoiceAttack (even if VoiceAttack is not listening). If you say what is in the input box before your command, VoiceAttack will listen to that command and go back to not listening. For instance, let's say you have a spoken command called, 'Attack', and VoiceAttack's listening is off. Let's also say that your listening override is, 'computer' (as shown). While listening is off, you can say, 'Attack' all day, and nothing will happen. If this option is on, you can say, 'Computer, Attack' and VoiceAttack will execute the, 'Attack' command (and then resume not listening).

'**Override global minimum confidence level**' allows you to set the minimum confidence level of recognized phrases. This overrides the value set globally on the, 'Options' screen. Note that you can override this value on individual commands as well. See, 'Options' screen for more information about confidence levels.

'Include commands from other profiles' will allow you to reference, or, 'include' the commands from any or all of your other profiles. This way, you can create common profiles filled with commands that can be shared among several other profiles. The profiles that you include can be arranged in priority, from highest to lowest. When duplicate-named commands are encountered, **the command in the profile with the higher priority will be retained**. For instance, let's say you have two profiles that you want to include: Profile A and Profile B. Profile A has been given a higher priority than Profile B (it's higher up on the list). Both profiles have a command called, 'Fire Weapons'. When you issue the command, 'Fire Weapons', the command from Profile A will be used, since Profile A has a higher priority.

Note: The included profiles in this set are higher priority than the Global Profiles selected from the Options screen (See, 'Global Profiles' on the Options screen). Also, the current, active profile always has the highest priority.

To edit the list of included profiles, click on the, '…' button to the right. This will bring up the, 'Include Profile Commands' screen. Use the controls on the right side of the screen to add, arrange and delete included profiles. Click, 'OK' when you are finished.

'Enable profile switching for the following windows or processes' - Checking this option will turn on automatic profile switching for this profile (this option works in conjunction with the option, 'Enable Auto Profile Switching' which is located on the Options screen). What automatic profile switching does is it allows you to specify one or more running applications to look out for. If a specified application is detected as the foreground window, VoiceAttack will automatically switch to this profile. To keep things as simple as possible, there is a text box that allows you to input the name of the window of the application that you want to look for. The input for this box is semicolon-delimited so you can associate your profile with more than one application. Since window titles vary depending on what you are doing, you can also add asterisks (*) as kind of a basic wildcard. If you put the asterisk at the end of the title, the search becomes, 'starts with' (for example, 'Notepad*'). If you put the asterisk at the beginning ('*Notepad'), the search becomes, 'ends with'. If you put an asterisk on both ends ('*Notepad*'), the search becomes, 'contains'. No asterisks ('Notepad') means a direct comparison (equals), and a single asterisk ('*') indicates that the profile is to be switched to if no other matches have been made (If VoiceAttack is the active window, the profile will not automatically switch (for obvious reasons)).

So, let's say you want your profile to automatically change when you switch over to either your desktop or Notepad. The desktop window name (oddly enough) is, 'Program Manager' (I know that's weird... there's some help about this down below). Notepad's window title *will change* depending on the document you are editing. Your input would look like this: 'Program Manager;*Notepad*' (without the quotes). That means VoiceAttack will look for a window titled, 'Program Manager' as well as any window that has a title that contains, 'Notepad'.

To help with finding out window titles, a new option has been added to the VoiceAttack Load Options screen. To get to this screen, simply start VoiceAttack while holding down CTRL + Shift. Select the option titled, 'Show window titles (requires 'Enable

Automatic Profile Switching' to be checked in the Options screen).' This will show the window titles in the log so you can see what VoiceAttack sees. Check out the Load Options screen for more details.

Update: VoiceAttack will now also allow you to search by process name in addition to window title. Just include the process name as you would the window title as indicated above, including wildcards. VoiceAttack will first search by window title, then by process name. Note that this is an advanced feature, so, if you have no idea what's being said here, just ignore this ;)

'**Send commands to this target**' - When VoiceAttack recognizes a command, and that command generates some type of input (keyboard key presses, mouse clicks), that command needs a place to send that input. You will most likely send that input to the Active Window (that is, the top-most, focused window – you know – the one you are looking at lol). You can also choose to have the input sent to an application with a specified window title, a named process (advanced) or even a window referenced by class name (way advanced). Enabling this option will make the current profile's commands target either the active window or another window/process that you specify to receive input. Lots of detail about process targets can be found in this document in the section titled, '**Application Focus (Process Target) Guide**'. Note that this setting can be overridden at the command level (see the 'Command Screen' for details).

To send input to the active window, choose, 'Active Window'. Whatever window that is currently active will receive input from the commands in this profile.

To send input to a specific window or process, choose the option next to the dropdown box. To see what windows are available, drop down the list. Choosing a window from the list will indicate to the commands in this profile that you want to send input to it. Note that this is a free input text box and you can modify your selection (as detailed below).

The value in the dropdown box can contain wildcards indicated by asterisks (*). This is handy when the title of the window changes. To indicate that the window title **contains** the value in the box, put an asterisk on each end. For instance, if you want to target any window that contains, 'Notepad' in the title, put, '*****Notepad*****' (without quotes) in the box. To indicate that the window title **starts with** the value in the box, put an asterisk at the end: 'Notepad*'. To indicate that the window title **ends with** the value in the box, put an asterisk at the beginning: '*Notepad'. The values are not case-sensitive. The first window found to match the criteria indicated will be selected.

Advanced: Note that you can also use process names as they appear in the Windows Task Manager. You can use wildcards the same as you do with the window titles. Window titles are checked first, and then the process names. See, 'Optimization note' below for an even finer level of control.

More advanced**:** If you need to find a window by window class name, you'll notice below in the, 'Optimization note' that you can prefix a, '+' (plus sign) to the beginning of the search term. Wildcards apply still if you need them. Note that this will be the class name of the window itself and not the class name of a child control. Again, this is a

highly-advanced feature that you may never ever use.

Optimization note - As indicated above, the, ‘target’ input box will accept the name of a window title or process name (or even class name), and checks for each of these items in that order. The reason for doing all of this in one go is for user simplicity (less user interface) as well as user assistance in locating their intended target (as oftentimes there is an overlap in naming). This works rather quickly in most cases, but in some situations the processing could be unnecessarily excessive. For instance, if you are looking for a process name that contains, ‘widget’, all window titles will be searched for, ‘widget’ first before the process names are searched. Again, this is a fast check, but it is unnecessary checking if you already know for sure that, ‘widget’ will only appear in a process name (and not a window’s title). In order to not add more user interface clutter and to provide those of you looking to eke out every bit of performance you can get, VoiceAttack now has character prefixes to limit searches to just the window title, process or class name.

If **'^'** is prepended to the target value, the target search is limited to only window titles.
If **'~'** is prepended, only process names are searched.
If **'+'** is prepended, only class names are searched.

These new prefixes can also be used in conjunction with wildcards.

For example, '^*notepad*' will indicate to only search window titles that contain, 'notepad'. If, ‘notepad’ is not found within a window title, the search will stop and not continue on to search processes or class names. ‘~widget*’ indicates that the search is to only look for process names that start with, ‘widget’. That means that no window titles or class names are searched. ‘+foo’ indicates that the search should only be for window class names that match, ‘foo’ (again, window titles and process names are not searched first which saves some time).

'Default Text-to-speech voice' - Selecting a value from this list will allow you to indicate a voice to be used profile-wide when, 'Default' is selected in a, 'Say Something with Text-to-Speech' action. Also, an attempt is made at using this selected voice when something goes wrong with the selected voice in the, ‘'Say Something with Text-to-Speech' action. You can select a voice from the list, or you can freely type in this box. The value typed in must resolve to an active voice. This can be a text variable, tokens or various combinations of literal text and tokens. Note that selecting, ‘None’ indicates that the Windows default voice will continue to be used.

Profile Options Exec Tab

'Execute a command each time a phrase is unrecognized' allows you to pick a command to run any time a phrase is not recognized. The selected command could be something as simple as playing a sound to calling a plugin function. Note that the '{CMD}' token takes on the unrecognized value so you can use it for processing (see the section on tokens near the end of this document).

'Execute a command each time this profile is loaded' allows you to pick a

command to execute when you switch to this profile (or when VoiceAttack is started and this profile is already selected). Again, your command can do simple stuff or it could initialize values for use later. If you need to disable this feature temporarily due to a problem (for example, loading another profile when your current profile loads... not a good idea), you can hold down CTRL and Shift when you launch VoiceAttack and choose the option, 'Disable profile initialization commands (this session only)' and click, 'OK'.

'**Execute a command each time this profile is unloaded**' allows you to pick a command that can be specified to execute immediately before the current, active profile is unloaded. A profile is *unloaded* when another profile is selected, or, when VoiceAttack is shutting down. Note that the profile that is selected to be loaded next will wait until the specified unload command completes, and any startup command in the selected profile will execute *after* the specified unload command. Note also that VA's shutdown will be delayed until the specified unload command completes executing. Tip: You can distinguish between the two types of unloading by checking the value of the {CMDACTION} token (see the section on tokens, later in this document). Tip: You can get some information about the profile that will load up after the current one unloads by checking out the, 'NEXTPROFILE' set of tokens (also later in this document).

'**Execute a command each time a dictation phrase is recognized**' allows you to choose a command that is run any time an entry is made into the dictation buffer. This could be useful for playing a sound, reading back what was last said or for plugin use.

Profile Options Hotkey Tab

Each of the items on this tab override what is available on the Options > Hotkeys screen. Select each item that you want to override by checking the appropriate box. Where a, '…' button is indicated, you can click it and be presented with a configuration screen (the base configuration is outlined in the Options screen). For example, if the Recognition Global Hotkey is, 'Ctrl + F5' on the Options screen (as a global setting), and we would like to have this particular profile use 'Alt + F1', we can override that value here. That means that for every other profile, you would press, 'Ctrl + F5' to toggle VoiceAttack's listening, but, when you are in this profile, you would use, 'Alt + F1'. This same principle is carried over into the, 'Mouse Click Recognition', 'Stop Command Hotkey' and the, 'Joystick Recognition Button' (See the Options page (Hotkeys tab) for more on what these features do).

Profile Options Advanced Tab

The '**Block potentially harmful profile actions'** option is useful for allowing you to inspect your profile before you put it into service. What this does is it works to prevent certain elements of a profile and its commands/actions from executing either intentionally or not intentionally. When this option is selected, the following elements are blocked from occurring:

- Profile switch actions (action not executed)
- Profile startup command execution (command not executed)
- Profile unload command execution (command not executed)
- Commands locked by author flag (command not executed)
- Run an application actions (action not executed)
- Stop process actions (action not executed)
- Inline function actions (action not executed)
- Plugin execution actions (action not executed)
- Plugin Command.Execute() function (command not executed)
- Unrecognized catch-all command execution (command not executed)
- Dictation recognized command execution (command not executed)
- Commands executed by command line (-command) (command not executed)

A message in the log will appear if any of the events above are invoked while this flag is set. Once you are satisfied with the contents of your profile, simply deselect this option.

NOTE: If you would like all profiles that are subsequently imported to have this option turned on by default, simply go to the Options screen and then to the, System/Advanced tab. On that screen you will find the option labeled, ‘Upon import, profiles will have, 'Block potentially harmful profile actions' selected’. Choose that option and each profile that is imported will have the, ‘Block potentially harmful profile actions' automatically set.

Now, back to the profile screen...

3 - New Command button

Click this button to add a new command to your profile (See 'Command Screen').

Edit Command button

Click to edit the currently selected command (same as pressing the Enter button on your keyboard or double-clicking a command in the command list) (See 'Command Screen').

Delete Command button

Click to delete the currently selected command (same as pressing the Delete button on your keyboard).

4 - Command list

This list shows all the commands that have been added by you for the currently selected profile. The first column shows the command name (the words that you will say into your microphone). In screenshot, the second column shows the category of your command (depending on your command list, you may also see the description and shortcut columns), and, the third column shows the actions that will be performed when VoiceAttack recognizes the command. For example, on the first line, the command, 'bags' is indicated. If you say the word, 'bags' into your microphone, VoiceAttack will press the, 'Left Shift' key, plus the, 'B' key. You can double-click a

command in this list to edit the command. Right-clicking on this list will allow you to add, edit and delete commands. **You can also copy and paste commands as well as copy commands to completely different profiles.**

5 - Import Commands button

Click this button to selectively import commands from previously-saved profiles (See, 'Importing Commands').

6 - List filter buttons

Expand/collapse multipart commands - This button toggles the view to show multipart commands (commands that have names separated by a semicolon) as one row or multiple rows.

Toggle Category Grouping - Use this button to group by category. Grouping by category will let you show/hide groups of commands that have the same category. Note that when the list is grouped by category, you can click on the group headers to expand or collapse that group. You can show and hide the group count by right-clicking on the header row and selecting the, 'Show Group Count' option. You can expand/collapse all groups If you hold down the CTRL key while expanding or collapsing a group. **Renaming categories for multiple commands can be done by right-clicking on the group header and selecting, 'Rename Category'.**

In this area, you will also see an indicator that shows how many commands are in your profile. If you have filters applied, you will see how many commands are available, plus how many are displayed. The tool tip you see when you hover over this information will display the number of derived commands that are created by things like dynamic commands or composite (prefix/suffix) commands.

Stop Commands Button - Clicking this button will stop any running commands (this works exactly like the, 'Stop All Commands' button on the Main screen).

List Filter Input Box - Start typing in this box and the command list will be filtered down to display any field that contains the text that is typed. Clear this box to remove the filter (this does not affect the underlying data).

List Filter Toggle Buttons - There are six buttons to quickly filter the list based on command state. You can toggle each filter by simply clicking the buttons (this does not affect the underlying data). The six filters are:
- Hide/show commands that have 'When I say' disabled. (S)
- Hide/show commands that have 'When I press keys' disabled. (K)
- Hide/show commands that have 'When I press button' disabled. (J)
- Hide/show commands that are prefixes. (P)
- Hide/show commands that are suffixes. (X)
- Hide/show commands that are full commands. (F)

Done button

No commands will be saved unless you hit the Done button. **NOTE**: New and edited commands ARE NOT AVAILABLE to the speech recognition engine until you press 'Done' (or, 'Apply'). There have been a lot of times when I have entered a new

command on the Profile screen and was puzzled by why it wasn't working when I would speak. It's because I never clicked the, 'Done' button. Happens a lot. Seriously... I am that dense :)

Apply button

Works just like the, ‘Done’ button, but does not close the screen.

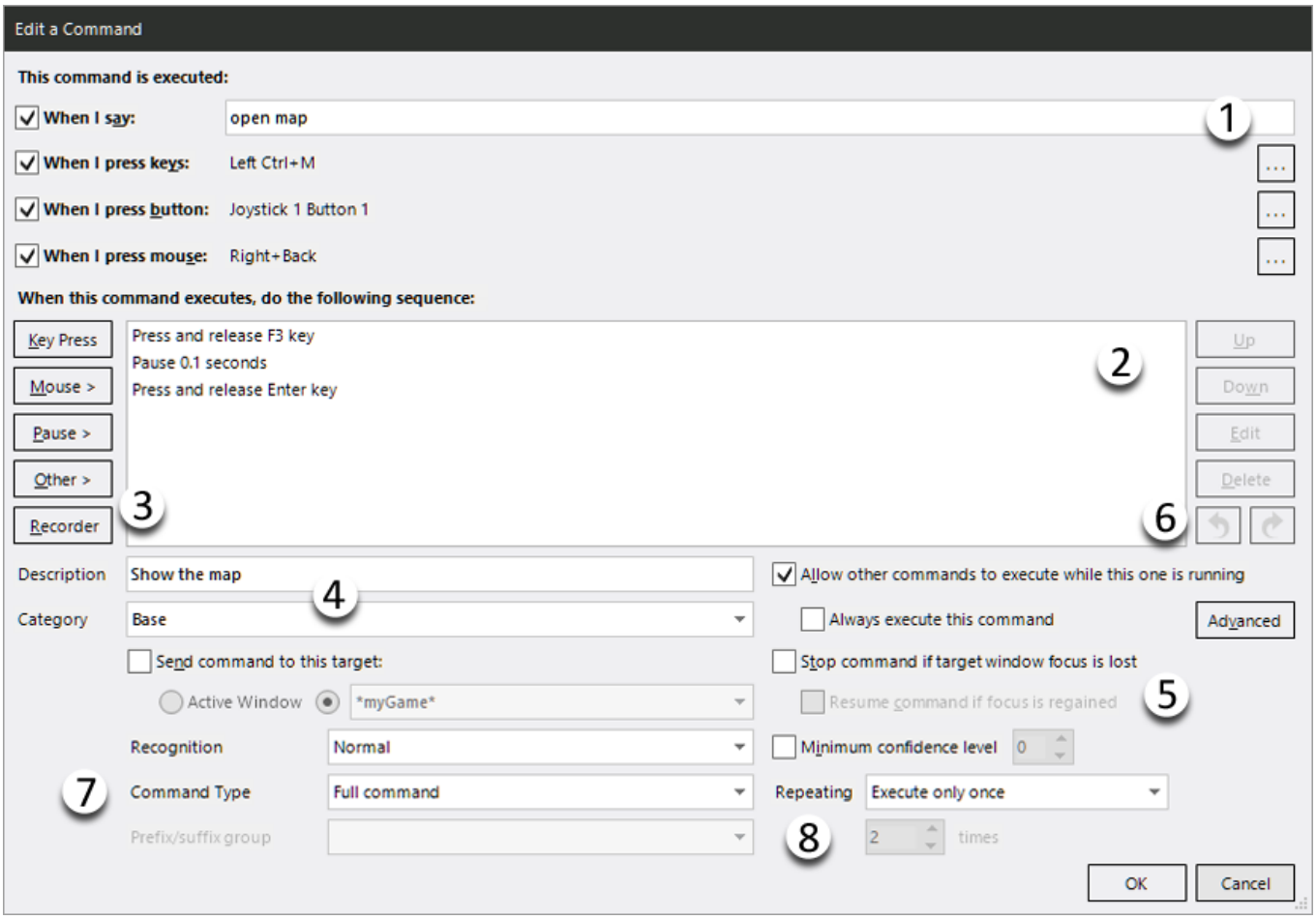
Cancel button

All changes to commands for this profile can be canceled by hitting the Cancel button.

Command Screen

A VoiceAttack command is basically a macro that performs a series of actions. A command can be executed with a spoken word or phrase, with the press of a keyboard shortcut or the press of joystick or mouse buttons. Simple commands can be run one at a time, while lengthy commands can be configured to run in the background (asynchronously).

The Command Screen is where you will add and edit commands and their actions that execute when VoiceAttack recognizes your spoken phrase or detects your keyboard shortcut / joystick button press (Note: If you edit multiple commands at once instead of just one, you will be presented with the, '**Command Multi-Edit**' screen. For more information, see the section regarding that screen later in this document.) For instance, you can add a command called, 'Help' and then actions that press and release the 'F1' key in your application.



In the above example, this command can be executed in four different ways. The first way is to speak the phrase, 'open map' into the microphone. The second is to press Ctrl + M on the keyboard. The third way is by pressing button 1 on joystick 1, and the fourth way is to press the right and back mouse buttons at the same time. VoiceAttack will react by sending the 'F3' key to your application, pausing briefly, and then send the 'Enter' key (your keen eye may have noticed that this command will run in the background (option group 5). More about that below). There is a lot going on here, but we'll go through each numbered section, one at a time.

1 - Command Input

The command input section up near the top is where you'll indicate how your command will be executed by user interaction. This can be with a spoken phrase, a keyboard hotkey/shortcut, a joystick button press, or by a particular mouse button click.

Checking the box labeled, 'When I say...' indicates to VoiceAttack that you want this command to be executed by speaking a word or phrase. In the example, we want to say the phrase, 'open map' into the microphone to get VoiceAttack to react.

Note that you *must* fill out the input box if this option is checked, and, what you put in the input box must be unique for the selected profile (in this case, you can only have one command with, 'open map' as the spoken phrase).
NOTE - VoiceAttack supports multiple phrases in the input box by separating the phrases with a semicolon ; For example, if you have three commands that do the same thing: Fire, Open Fire and Fire Weapons, instead of adding three separate commands, you can add one command like this: 'Fire;Open Fire;Fire Weapons' and VoiceAttack will execute the commands all the same way.

Dynamic command sections

Dynamic sections allow you to specify a part of your command that may vary. Sometimes you may want to say, 'Hello computer' and sometimes you may want to say, 'Greetings computer' and execute the same command. To indicate that you want to use a dynamic section, enclose the section in square brackets: [ ], with each element separated by a semicolon. Your command may look something like this:

[Hello;Greetings]computer

In this case, you can say, 'Hello computer' and 'Greetings computer' and the command will be executed.

Note that multipart commands are also still separated with a semicolon (as demonstrated by adding, 'Hi' to the end):

[Greetings;Hello]computer;Hi

With this example, to execute the command, you can say:

Greetings computer
Hello computer
Hi

The dynamic sections don't have to just be at the beginning. They can be anywhere in the command. Also, as a side-effect, **if you put a semicolon on at the end of the selections, it makes the section optional**:

[Greetings;Hello]computer[how are you;]

You can say the following to execute the command:

Greetings computer how are you
Hello computer how are you
Greetings computer
Hello computer

Note that there is a semicolon after 'how are you' to indicate that the entire section is optional.

Something to consider when using this feature is that you can create a lot of permutations from very few words. Use with care :)

Dynamic command sections can also contain **numeric ranges.** This is kind of an advanced feature that is not very useful on its own, but when used in conjunction with other features (such as the {TXTNUM} token (see, 'Tokens' section) and Quick Input), it can be used to consolidate large numbers of commands into just a few.
To indicate a numeric range in a dynamic section, just include the minimum and maximum values separated by an ellipsis (..). For example, let's say you have 100 racers in a race game. Instead of creating 100 separate commands to eject racers, you can have a single command, 'eject car [1..100]'. With this example, you can say, 'eject car 1', 'eject car 2', 'eject car 99', etc.

An additional option for numeric ranges for dynamic command sections is a **multiplier**. This will allow you to, 'step' the numbers in your range by a specified value. To indicate a multiplier, just include the value with your range like this: [1..5,10]. Just like above, that's the minimum value and maximum value separated by an ellipsis, then a comma, then the multiplier value. [1..5,10] will yield 10, 20, 30, 40, 50. [5..10,20] will yield 100, 120, 140, 160, 180, 200.

Advanced: If you would like to retrieve the individual portions of a command that contains dynamic command sections, see the, '{CMDSEGMENT:}' token later in this document.

Wildcards

There is a somewhat unsupported* feature in VoiceAttack's 'When I say' feature. You can use, 'wildcards' around the phrases to indicate, 'contains', 'starts with' and 'ends with'.

So, let's say you have a spoken phrase 'attack'. Let's also say that you want to execute your command if the word, 'attack' is included in any spoken phrase. To indicate to VoiceAttack that you want the 'attack' command to execute any time it is contained in a phrase, you simply put asterisks around the phrase, like so: *attack*.

If you want to indicate that you want the 'attack' command to execute if the spoken phrase starts with the word, 'attack', just put an asterisk at the end, like so: attack*. This way, you can say, 'attack the enemy' and VoiceAttack will execute the 'attack'

command. If you say, 'I would like to attack the enemy', the 'attack' command will not be executed, since the word, 'attack' is not at the start of the phrase.

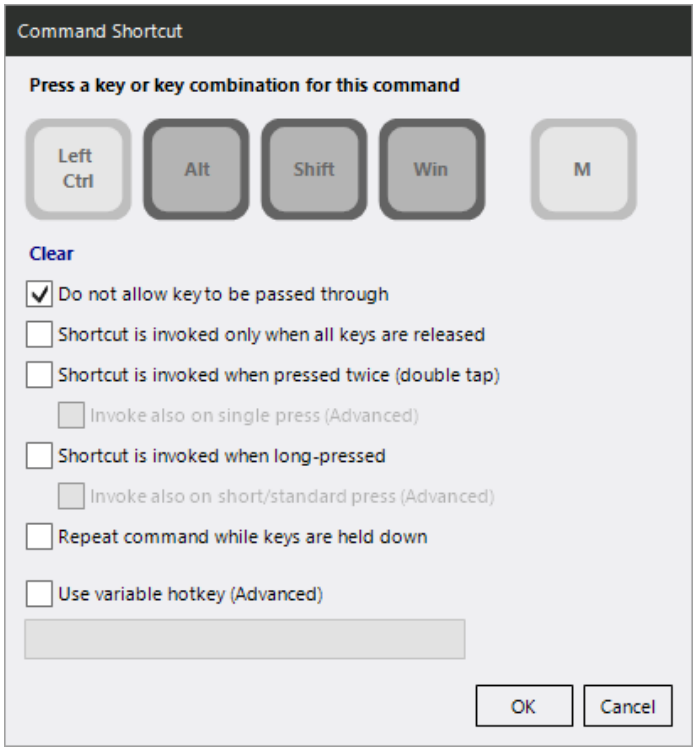
On a similar note, if you only want, 'attack' to be executed if the word, 'attack' is at the end of the phrase, put the asterisk at the beginning, like so: *attack.

VoiceAttack will execute all of the commands in which the wildcards apply. So, if you have '*rocket*' and '*ship*' and '*attack*' as commands, and you happen to say, 'I would like to attack the ship with my rockets', VoiceAttack will attempt to execute 'attack', then, 'ship' and then 'rocket' in that order (the order in which they are spoken, but due to the asynchronous nature of this type of situation, the order cannot be guaranteed (use with caution).

Commands will not repeat with wildcards. If you have commands, '*rocket*' and, '*ship*' and, '*rocket ship*' and you say, 'I want to take a ride in my rocket ship', VoiceAttack will execute the command, 'rocket ship' and not 'rocket' and 'ship'. Also, if you say, 'rocket rocket rocket rocket rocket rocket rocket rocket rocket', the 'rocket' command will only be executed once.

* The reason it is 'somewhat unsupported' is basically because it is not a terribly reliable feature and was added as an attempt to give a little bit more flexibility, especially in the areas of immersion. Your mileage may vary. Good luck, captain!

Checking the box labeled, '**When I press keys**' indicates to VoiceAttack that you want to execute this command when pressing a keyboard shortcut. In this example, the keyboard shortcut is Left Ctrl + M. You can assign the keyboard shortcut by clicking on the '…' button to the right. You will be presented with the screen below:



This is the, 'Command Shortcut' screen. It allows you to choose the key / key combo to assign to your macro.

The, '**Do not allow key to be passed through**' option prevents the main key (non-modifier) from being passed through to the application. For example, if your hotkey is F1 and this option is selected, VoiceAttack will respond to the F1 key press and then prevent any other application from receiving this key press (for this example, if F1 is being handled by VoiceAttack, you will not be able to use the F1 key in other applications while VoiceAttack is running. If you rely on F1 to bring up, 'Help', then, you'll have to pick another key).

The, '**Shortcut is invoked only when all keys are released**' option allows you to indicate that the macro will only execute once all keys in the combo are up. This allows you a greater level of flexibility and control (such as having a macro that executes on key down and a separate macro that occurs only on key up). Also, this is the way that VoiceAttack can keep hotkey shortcuts from stepping on each other when some of the keys are involved in different commands. For instance, if there is a command that is executed by pressing, 'ALT + X', and another command that executes by pressing, 'X', setting both shortcuts to work on key release will keep both from executing at the same time.

'**Shortcut is invoked when pressed twice (double tap)**' indicates that the command will be executed if the key or key combination is pressed twice (double tap). If the keys are only pressed once, the command will not be executed. You must perform the double tap within a specified amount of time in order for it to be considered a double tap and subsequently execute your command. The threshold/timeout for a double tap can be adjusted by going to the Options screen and changing the, '**Keyboard shortcut double tap threshold (ms)**' value on the Hotkeys tab. The default threshold value is 300 milliseconds. Decrease this number if your key presses are pretty fast, and increase if you would like a little more time to perform the double tap. Note that if you designate a key/key combination to a double tap as well as a long tap, conflicts will occur.

Note that you can have other commands with the same key combination that can execute on a single press (single tap). The double tap command will override the corresponding single tap command. A single tap will execute if it is pressed once and the threshold for a double tap is exceeded. For example, let's say you have a double tap command (Command A) assigned to the F12 key, and you also have another command (Command B) that uses the F12 key that works on a single key press. If you press F12 twice quickly, Command A will execute, but Command B will not (Command A's double tap overrides Command B's single tap). If you just press F12 once and you wait more than 300ms (the default threshold), it will be assumed that you meant to issue a single tap and Command B will be executed.

Another added advanced feature for double tap (if it's not unclear enough lol) is the, '**Invoke also on single press (Advanced)**' option. This indicates that a command that is designated to execute with a double tap can also be executed as a single tap. This is so you can keep a double tap and a single tap for a given key combination within one command. For execution flow, the way you can tell if the command is executed as

a double tap (versus a single tap) is to check the value of the, '{CMDDOUBLETAPINVOKED}' token. If the value rendered is, '1', the command was executed as a double tap. If the value rendered is, '0', the command was executed as a single tap (see VoiceAttack's token reference later in this document). Note that this option is not available when long press is enabled (due to conflicts). Again, this is an advanced feature. It's probably not a feature you may ever use, but it's there if you need it.

'**Shortcut is invoked when long-pressed**' indicates that the command will be executed if the key or key combination is held for a longer period of time. If the keys are pressed in a short period of time, the command will not be executed. You must hold down the key for a specified amount of time in order for it to be considered a long press and subsequently execute your command. The threshold time value for a long press can be adjusted by going to the Options screen and changing the, '**Keyboard shortcut long press threshold (ms)**' value on the Hotkeys tab. The default threshold value is 700 milliseconds. Increasing this value will require you to hold a key down longer to indicate a long press. Decrease to reduce the time needed.

Note that you can have other commands with the same key combination that can execute on a short/standard press. The long-pressed command will override the corresponding short/standard command. A short/standard press will execute if it is pressed quickly. For example, let's say you have a long press command (Command A) assigned to the F12 key, and you also have another command (Command B) that uses the F12 key that works on a short/standard key press. If you press F12 down for a little longer, Command A will execute, but Command B will not (Command A's long press overrides Command B's short/standard press). If you just press F12 once quickly (quicker than the default of 700ms - the default threshold), it will be assumed that you meant to issue a short/standard press and Command B will be executed.

Another added advanced feature for long press (if it's not unclear enough lol) is the, '**Invoke also on short/standard press (Advanced)**' option. This indicates that a command that is designated to execute with a long press can also be executed as a short/standard press. This is so you can keep a long press and a short/standard press for a given key combination within one command. For execution flow, the way you can tell if the command is executed as a long press (versus a short/standard press) is to check the value of the, '{CMDLONGPRESSINVOKED}' token. If the value rendered is, '1', the command was executed as a long press. If the value rendered is, '0', the command was executed as a short/standard press (see VoiceAttack's token reference later in this document). Note that this option is not available when double tap is enabled (due to conflicts). Again, this is an advanced feature. It's probably not a feature you may ever use, but it's there if you need it. Note also that if you designate a key/key combination to a long tap as well as a double tap, conflicts will occur.

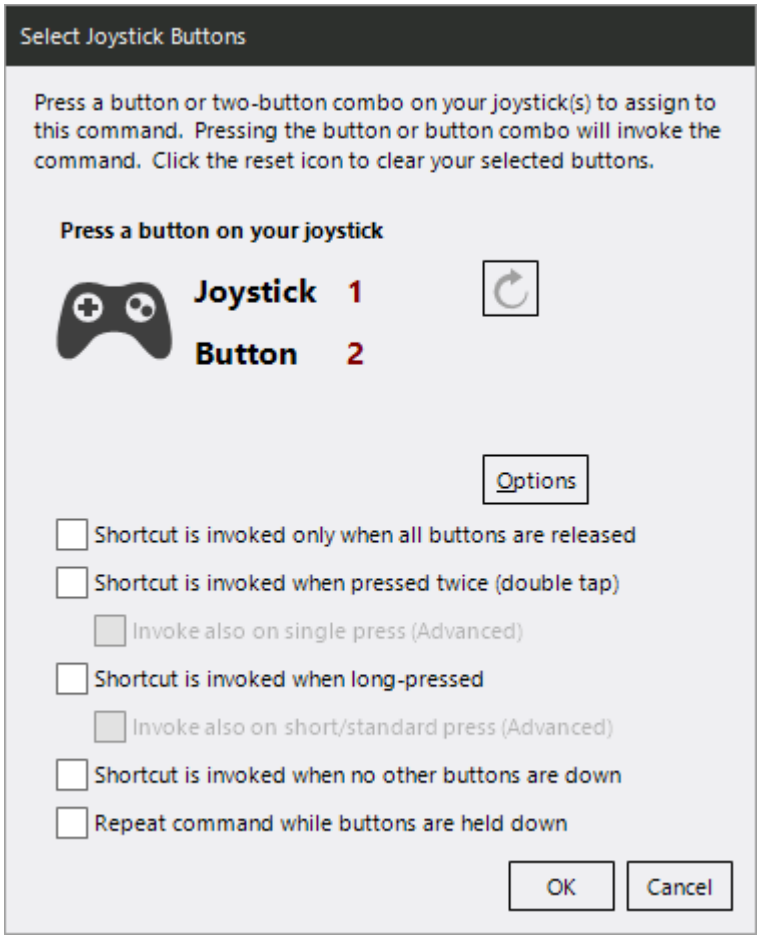
The '**Repeat command while keys are held down**' option will allow the invoked command to continuously repeat for as long as the selected keyboard keys are held down.

The '**Use variable hotkey (Advanced)**' option is provided for you to be able to use a hotkey that is indicated by the contents of a text variable. This is so that a

hotkey/hotkey combo can be assigned to your command that is not known until the profile is actually loaded or is running. To make this feature work properly for you, there are a few items to keep in mind. To turn on the variable hotkey for a command, make sure the box is checked and simply put the name of the text variable to use in the provided input box. The previously-set **text variable must contain properly-notated text in order to work**. The good news is that the notation is (almost) exactly the same as what you will find in the Quick Input action and for variable keypresses. So, for instance, if your desired hotkey (for now) is ALT + L, set a text variable's value to '[ALT]L' (no quotes).
Note that the, 'L' does not have brackets. Keys with a single-character identifier (A-Z, +, ß, ç, etc.) do not need brackets. Special keys, such as Enter, Shift, Alt, F12, etc. will require brackets (see the section titled, 'Quick Input, Variable Keypress and Hotkey Key Indicators' for all the possible key indicators). Note also that there is no space between [ALT] and L. Spaces are actually picked up as hotkeys here, so if there is a space, the space bar will be monitored for the given command.

Something to understand is that the value in your text variable can change at any time, and the hotkey monitoring process that VoiceAttack uses is optimized so that **you must explicitly refresh the hotkeys** when a variable value is changed. In order to refresh the hotkeys that VoiceAttack is monitoring, you simply execute a, '**Refresh Variable Hotkeys**' action (see the section about this action later in this document). Note that only global and profile-scoped variables will work with this feature (command-scoped variables are inaccessible).

Checking the box labeled, '**When I press button**' indicates to VoiceAttack that you want to execute this command when pressing a joystick button, or a button combination. Note: Setting up joystick support is presented in more detail in the Options screen under the heading, 'Joystick Options'.) In this example, the selected button is the second button on joystick 1. Note that you can use up to two buttons to create a button combo (the buttons can even be on different sticks if you want). You can assign the joystick button for this command by clicking on the '…' button to the right. You will be presented with the screen below:



This is the 'Select Joystick Buttons' screen. If your joysticks (up to two joysticks are supported) are plugged in and set up, you can press a button and it will be detected here. To clear anything that you've done on this screen, click the, 'reset' button in the top-right. The option, '**Shortcut is invoked only when all buttons are released**' will make the command only execute when all the buttons involved are let go. Note that any superseded shortcuts will not be executed when using this option. That means if you have a shortcut that works when the, 'A' button is released as well as a shortcut that works when, 'A + B' are released, only the shortcut for 'A + B' will be executed. The option, '**Shortcut is invoked when no other buttons are down**' will prevent the command from executing if the buttons indicated are not exclusively involved. For example, if you have a command that is executed when button 'A' is pressed and this option is selected, if any other button is down when, 'A' is pressed, the command will not be executed. If you need to change the availability or assignment of your joysticks, you can access the joystick options screen from here by clicking the, 'Options' button.

'**Shortcut is invoked when pressed twice (double tap)**' indicates that the command will be executed if the button or button combination is pressed twice (double tap). If the buttons are only pressed once, the command will not be executed. You must perform the double tap within a specified amount of time in order for it to be considered a double tap and subsequently execute your command. The threshold/timeout for a double tap can be adjusted by going to the Options screen and changing the, '**Joystick button shortcut double tap threshold (ms)**' value on the Hotkeys tab. The

default threshold value is 300 milliseconds. Decrease this number if your button presses are pretty fast, and increase if you would like a little more time to perform the double tap.

Note that you can have other commands with the same button combination that can execute on a single press (single tap). The double tap command will override the corresponding single tap command. A single tap will execute if it is pressed once and the threshold for a double tap is exceeded. For example, let's say you have a double tap command (Command A) assigned to Joystick 1, Button 1, and you also have another command (Command B) that uses Joystick 1, Button 1, that works on a single press. If you press Button 1 twice quickly, Command A will execute, but Command B will not (Command A's double tap overrides Command B's single tap). If you just press Button 1 once and you wait more than 300ms (the default threshold), it will be assumed that you meant to issue a single tap and Command B will be executed.

Another added advanced feature for double tap (if it's not unclear enough lol) is the, '**Invoke also on single press (Advanced)**' option. This indicates that a command that is designated to execute with a double tap can also be executed as a single tap. This is so you can keep a double tap and a single tap for a given button combination within one command. For execution flow, the way you can tell if the command is executed as a double tap (versus a single tap) is to check the value of the, '{CMDDOUBLETAPINVOKED}' token. If the value rendered is, '1', the command was executed as a double tap. If the value rendered is, '0', the command was executed as a single tap (see VoiceAttack's token reference later in this document). Again, this is an advanced feature. It's probably not a feature you may ever use, but it's there if you need it.

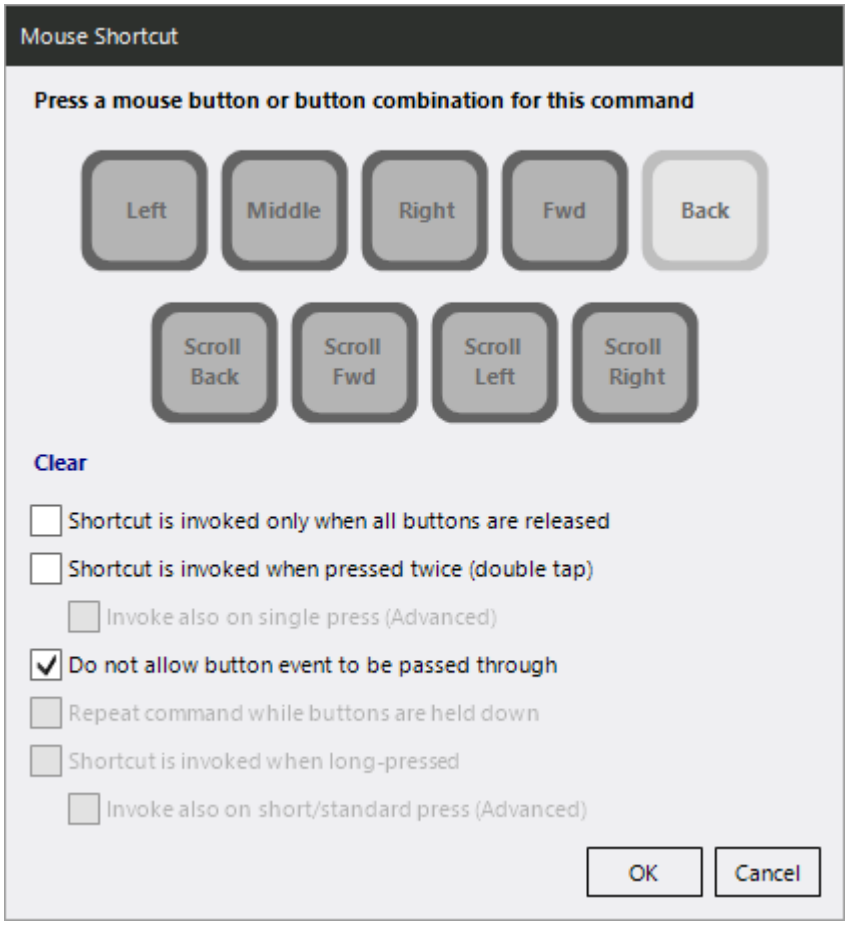
'**Shortcut is invoked when long-pressed**' indicates that the command will be executed if the button or button combination is held for a longer period of time. If the buttons are pressed in a short period of time, the command will not be executed. You must hold down the button for a specified amount of time in order for it to be considered a long press and subsequently execute your command. The threshold time value for a long press can be adjusted by going to the Options screen and changing the, '**Joystick button long press threshold (ms)**' value on the Hotkeys tab. The default threshold value is 700 milliseconds. Increasing this value will require you to hold a button down longer to indicate a long press. Decrease to reduce the time needed.

Note that you can have other commands with the same button combination that can execute on a short/standard press. The long-pressed command will override the corresponding short/standard command. A short/standard press will execute if it is pressed quickly. For example, let's say you have a long press command (Command A) assigned to Joystick 1 button 1, and you also have another command (Command B) that uses Joystick 1 button 1 that works on a short/standard key press. If you press Joystick 1 button 1 down for a little longer, Command A will execute, but Command B will not (Command A's long press overrides Command B's short/standard press). If you just press Joystick 1 button 1 once quickly (quicker than the default of 700ms - the default threshold), it will be assumed that you meant to issue a short/standard press and Command B will be executed.

Another added advanced feature for long press (if it's not unclear enough lol) is the, '**Invoke also on short/standard press (Advanced)**' option. This indicates that a command that is designated to execute with a long press can also be executed as a short/standard press. This is so you can keep a long press and a short/standard press for a given button combination within one command. For execution flow, the way you can tell if the command is executed as a long press (versus a short/standard press) is to check the value of the, '{CMDLONGPRESSINVOKED}' token. If the value rendered is, '1', the command was executed as a long press. If the value rendered is, '0', the command was executed as a short/standard press (see VoiceAttack's token reference later in this document). Note that this option is not available when double tap is enabled (due to conflicts). Again, this is an advanced feature. It's probably not a feature you may ever use, but it's there if you need it.

The '**Repeat command while buttons are held down**' option will allow the invoked command to repeat for as long as the selected joystick buttons are held down.

Checking the box labeled, '**When I press mouse**' indicates to VoiceAttack that you want to execute this command when pressing a mouse button (or button combination) or scrolling with the mouse scroll wheel. In this example, the, 'Back' button is selected. Note that you can use any combination of the five standard mouse buttons, and any single scroll wheel event. You can assign the mouse button for this command by clicking on the '…' button to the right. You will be presented with the screen below:



This is the 'Mouse Shortcut' screen. To clear anything that you've done on this screen, click the, 'clear' link. The option, '**Shortcut is invoked only when all buttons are released**' will make the command only execute when all the buttons involved are let go. Note that any superseded shortcuts will not be executed when using this option. That means if you have a shortcut that works when the, 'Back' button is released as well as a shortcut that works when, 'Back + Forward' are released, only the shortcut for 'Back + Forward' will be executed. The, '**Do not allow button event to be passed through**' option prevents the button-down event for the indicated buttons from being passed through to the focused application. For example, if your selected button is, 'Back' and this option is selected, VoiceAttack will respond to 'Back' button press and then prevent any other application from receiving this button press (for this example, if 'Back' is being handled by VoiceAttack, you will not be able to use the, 'Back' button in other applications while VoiceAttack is running. If you rely on, 'Back' to go back in your browser, then, you will have to pick another button). Note that this option affects other options' availability. Note also that checking this option affects the rendered value of several {STATE} tokens (See '{STATE_LEFTMOUSEBUTTON}, {STATE_RIGHTMOUSEBUTTON}, {STATE_MIDDLEMOUSEBUTTON}, {STATE_FORWARDMOUSEBUTTON}, {STATE_BACKMOUSEBUTTON} and {STATE_ANYMOUSEDOWN} for more info).

'**Shortcut is invoked when pressed twice (double tap)**' indicates that the command will be executed if the button or button combination is pressed twice (double tap). If the buttons are only pressed once, the command will not be executed. You must perform the double tap within a specified amount of time in order for it to be considered a double tap and subsequently execute your command. The threshold/timeout for a double tap can be adjusted by going to the Options screen and changing the, '**Mouse shortcut double tap threshold (ms)**' value on the Hotkeys tab. The default threshold value is 300 milliseconds. Decrease this number if your button presses are pretty fast, and increase if you would like a little more time to perform the double tap.

Note that you can have other commands with the same button combination that can execute on a single press (single tap). The double tap command will override the corresponding single tap command. A single tap will execute if it is pressed once and the threshold for a double tap is exceeded. For example, let's say you have a double tap command (Command A) assigned to the Back button, and you also have another command (Command B) that uses the Back button that works on a single press. If you press Back twice quickly, Command A will execute, but Command B will not (Command A's double tap overrides Command B's single tap). If you just press Back once and you wait more than 300ms (the default threshold), it will be assumed that you meant to issue a single tap and Command B will be executed.

Another added advanced feature for double tap (if it's not unclear enough lol) is the, '**Invoke also on single press (Advanced)**' option. This indicates that a command that is designated to execute with a double tap can also be executed as a single tap. This is so you can keep a double tap and a single tap for a given button combination within one command. For execution flow, the way you can tell if the command is executed as a double tap (versus a single tap) is to check the value of the, '{CMDDOUBLETAPINVOKED}' token. If the value rendered is, '1', the command was

executed as a double tap. If the value rendered is, ‘0’, the command was executed as a single tap (see VoiceAttack’s token reference later in this document). Again, this is an advanced feature. It’s probably not a feature you may ever use, but it’s there if you need it.

The ‘**Repeat command while buttons are held down**’ option will allow the invoked command to repeat for as long as the selected mouse buttons are held down.
Note: This option is only available when the, ‘Do not allow button events to be passed through’ option is NOT checked.

Note: ‘Shortcut is invoked only when all the buttons are released’ is not applicable to the scroll wheel events as they do not have a, ‘down’ or ‘up’ state.

Note: Scroll wheel events are triggered for each, ‘click’, so, when scrolling forward or back, the command will be triggered each time the mouse wheel, ‘clicks’ in either direction. For left and right, holding the mouse down in either direction will cause the command to repeat as long as the wheel is held.

‘**Shortcut is invoked when long-pressed**’ indicates that the command will be executed if the button or button combination is held for a longer period of time. If the buttons are pressed in a short period of time, the command will not be executed. You must hold down the button for a specified amount of time in order for it to be considered a long press and subsequently execute your command. The threshold time value for a long press can be adjusted by going to the Options screen and changing the, ‘**Mouse button long press threshold (ms)**’ value on the Hotkeys tab. The default threshold value is 700 milliseconds. Increasing this value will require you to hold a button down longer to indicate a long press. Decrease to reduce the time needed. Note that this feature is not available for scroll wheel actions.

Note that you can have other commands with the same button combination that can execute on a short/standard press. The long-pressed command will override the corresponding short/standard command. A short/standard press will execute if it is pressed quickly. For example, let’s say you have a long press command (Command A) assigned to the Back button, and you also have another command (Command B) that uses the Back button that works on a short/standard key press. If you press Back down for a little longer, Command A will execute, but Command B will not (Command A’s long press overrides Command B’s short/standard press). If you just press Back once quickly (quicker than the default of 700ms - the default threshold), it will be assumed that you meant to issue a short/standard press and Command B will be executed. Note that this feature is not available when, 'Do not allow button event to be passed through’ has been checked.

Another added advanced feature for long press (if it’s not unclear enough lol) is the, ‘**Invoke also on short/standard press (Advanced)**’ option. This indicates that a command that is designated to execute with a long press can also be executed as a short/standard press. This is so you can keep a long press and a short/standard press for a given button combination within one command. For execution flow, the way you can tell if the command is executed as a long press (versus a short/standard press) is to check the value of the, ‘{CMDLONGPRESSINVOKED}’ token. If the value rendered

is, ‘1’, the command was executed as a long press. If the value rendered is, ‘0’, the command was executed as a short/standard press (see VoiceAttack’s token reference later in this document). Note that this option is not available when double tap is enabled (due to conflicts). Again, this is an advanced feature. It’s probably not a feature you may ever use, but it’s there if you need it.

2 - Command Macro Action List

This is a list of all of the actions that will be performed, in order. In the example, we are sending a series of keyboard key presses with a pause in between. The items in the list can be key presses, mouse clicks, pauses, application launches, etc. You can double-click on any item in the list to edit that item. Note that you can change the order of the items in the list by moving them up and down.

3 - Actions

Add a Key Press action button
Click this button to add a keyboard key press to the command action sequence (See 'Key Press Screen'). You will probably be using this the most often.

Add a Mouse action button
Click this button to add a mouse action (such as moving the mouse, clicking a mouse button) to the command action sequence (See 'Mouse Action Screen').

Add a Pause action button
Click this button to add a timed pause to the command action sequence (See 'Pause Screen' and ‘Variable Pause Screen’).

Add a miscellaneous, 'Other' action button
Click this button if you want to launch or close an application (among other things). (See 'Other Stuff Screen'). Lots of fun items in here.

Recorder button
Click this button to bring up the 'Key Press Recorder' screen. This screen will capture key presses as you perform them for insertion into your macro. This will make input easier for a series of keys (instead of adding one at a time). (See, 'Record Keypress Screen').

4 - Command Organization

Description
This is where you can give a description of your command. **Note:** The description column will not show up in the Commands list of the Profile screen unless you actually have at least one command with a description.

Category
This allows you to organize your commands in a simple fashion. You can type in a new category in the box, or, you may drop down the list to select a category that

already exists in your profile. **Note:** The category column will not show up in the Commands list unless you actually have at least one command with a category.

5 – Command Execution Options

'Allow other commands to execute while this one is running' option.

When VoiceAttack recognizes a command, it is executed in one of two ways. One way is synchronous (the option is unselected). That is, when the command is executing, subsequent, recognized commands must wait until the command is finished before they execute. This is useful when you want the command to do its work without other commands interfering. When other commands try to execute, they can be canceled, or they can wait and then execute. The setting, 'Cancel blocked commands' on the options page will allow you to choose what to do with the commands that are blocked. If you choose to cancel blocked commands, the commands are simply ignored. If you choose to let the blocked commands wait, when the blocking command is finished and releases the block, all the waiting commands will then execute. Since the waiting commands are not queued and waiting within their own threads, these commands will execute simultaneously when unblocked, so use with caution. Note that calling a command that has this option set does not affect commands that are already running.

The second way to execute a command is asynchronously (the option is selected). That is, when the command is executing, VoiceAttack will continue to execute subsequent, recognized commands. This is the default behavior of VoiceAttack. Note that this option is applicable only to the root executed command. Commands executed as sub-commands (see 'Execute Another Command' action) will follow the root command's setting. Please exercise caution when using this option, since key presses can interfere with each other. Note that you can click the 'Stop Commands' button on the Main Screen or add a command to stop all processing commands (see, 'Other Stuff' screen). This is basically a panic button that indicates to all running macros that they need to stop processing.

'Always execute this command' - This option indicates that the command is to execute regardless if another command is blocking it (see the previous option, 'Allow other commands to execute while this one is running'). Use this option when you know for sure that your command will not interfere with other commands when it is executed. Note that this option is only available when 'Allow other commands to execute while this one is running' is enabled.

'Stop command if target window focus is lost' - Enabling this option will make this command stop if the focused window loses focus. The focused window is the window that has the focus when the command is first executed. One handy use for this feature is if you have a command that continually loops and interacts with a certain application. If you click out of the window and lose focus, you probably do not want processing to continue on the newly active window (especially if keys are being pressed). This feature also has an option called, 'Resume command if focus regained'. What this does is attempt to continue the command if you focus the original window.

'Send command to this target' - Enabling this option will make this command target either the active window or another window/process that you specify to receive input. This will override the target indicated at the profile level (if specified – see, 'Profile Options' for more information).
To send input to the active window, choose, 'Active Window'. Choosing the active window at this level is handy when the profile or global settings are directed to a specific application. Whatever window that is currently active will receive input from this command.

To send input to a specific window or process, choose the option next to the dropdown box. To see what windows are available, drop down the list. Choosing a window from the list will indicate to the command that you want to send input to it. Note that this is a free input text box and you can modify your selection (as detailed below).

The value in the dropdown box can contain wildcards indicated by asterisks (*). This is handy when the title of the window changes. To indicate that the window title **contains** the value in the box, put an asterisk on each end. For instance, if you want to target any window that contains, 'Notepad' in the title, put, '*Notepad*' (without quotes) in the box. To indicate that the window title **starts with** the value in the box, put an asterisk at the end: 'Notepad*'. To indicate that the window title **ends with** the value in the box, put an asterisk at the beginning: '*Notepad'. The values are not case-sensitive. The first window found to match the criteria indicated will be selected. This option will also accept any number of tokens and will work if the variables rendered by the tokens are set prior to the command being executed.

Advanced: Note that you can also use process names as they appear in the Windows Task Manager. You can use wildcards the same as you do with the window titles. Window titles are checked first, and then the process names.

More advanced: If you need to find a window by window class name, you'll notice below in the, 'Optimization note' that you can prefix a, '+' (plus sign) to the beginning of the search term. Wildcards apply still if you need them. Note that this will be the class name of the window itself and not the class name of a child control. Again, this is a highly-advanced feature that you may never ever use.

Optimization note - As indicated above, the, 'target' input box will accept the name of a window title or process name (or even class name), and checks for each of these items in that order. The reason for doing all of this in one go is for user simplicity (less user interface) as well as user assistance in locating their intended target (as oftentimes there is an overlap in naming). This works rather quickly in most cases, but in some situations the processing could be unnecessarily excessive. For instance, if you are looking for a process name that contains, 'widget', all window titles will be searched for, 'widget' first before the process names are searched. Again, this is a fast check, but it is unnecessary checking if you already know for sure that, 'widget' will only appear in a process name (and not a window's title). In order to not add more user interface clutter and to provide those of you looking to eke out every bit of performance you can get, VoiceAttack now has character prefixes to limit searches to just the window title, process or class name.

If '**^**' is prepended to the target value, the target search is limited to only window titles.
If '**~**' is prepended, only process names are searched.
If '**+**' is prepended, only class names are searched.

These new prefixes can also be used in conjunction with wildcards.

For example, '^*notepad*' will indicate to only search window titles that contain, 'notepad'. If, 'notepad' is not found within a window title, the search will stop and not continue on to search processes or class names. '~widget*' indicates that the search is to only look for process names that start with, 'widget'. That means that no window titles or class names are searched. '+foo' indicates that the search should only be for window class names that match, 'foo' (again, window titles and process names are not searched first which saves some time).

Lots more information about process targets is available in this document in the section titled, '**Application Focus (Process Target) Guide**'.

'Minimum Confidence Level' allows you to specify to VoiceAttack what the minimum recognition confidence level must be in order to execute this command. This value overrides the values set at global level as well as the profile level. See the, 'Options' page for more information about the handy confidence feature.

'Recognition' – this is currently an experimental feature that will allow you to possibly speed up how fast your spoken command is recognized by the speech engine.
Normally, your speech engine waits for you to briefly finish speaking before making a final decision about what you just said (and subsequently execute a command if it finds a match). This behavior of the speech engine is represented by the, '**Normal**' setting for Recognition, and is the default selection (you know – how VA has always behaved).

Now for the good stuff… Your speech engine is constantly listening to you, and as the speech engine is listening, it is attempting to contextually piece together what you are saying based on the spoken command phrases you've created in your profile. As the speech engine gathers up what can be constituted as one of your spoken phrases within continuous speech, you have the option to execute your command at that point in time rather than wait for the speech engine that is also waiting for you to stop speaking. This can shave off a few milliseconds in your speech event. Select the, '**Continuous Speech**' option to attempt to execute your command as part of continuous speech. The, '**Restricted Continuous Speech**' option works exactly like the, 'Continuous Speech' option, but the phrase to be recognized must be at the beginning of the speech event. The, 'Restricted' option is the one that I prefer the most, as it allows me to issue commands by themselves (as we've always done) as well as bypass the speech event delay somewhat. So, for example, let's say you have the spoken phrase, 'fire weapons'. With, 'Normal' selected, if you say, 'Ok fire weapons', the speech engine accepts, 'Ok fire weapons' as the phrase you are trying to execute (this is after the brief pause that we are all familiar with). The phrase, 'Ok fire weapons' is not found and you'll be met with, 'Unrecognized: Ok fire weapons' (of course). If you choose, 'Continuous Speech', when the speech engine encounters, 'Ok fire weapons', it sees, 'fire weapons' and doesn't wait for you to finish speaking before

executing the, ‘fire weapons’ command. This is great, but if you don’t like having the chance of your weapons firing in the middle of a sentence, you will want to give, ‘Restricted Continuous Speech’ a try. The spoken phrase, ‘I like turtles and fire weapons’ will not be picked up as a recognized command because, ‘fire weapons’ was not spoken at the start of the speech event. The spoken phrase, ‘fire weapons’ will get picked up as a recognized command (of course) as well as, ‘fire weapons and stuff’ (again, since, ‘fire weapons’ is at the start of the phrase (speech event)).

So, what’s the catch? Well, you knew there would be some kind of catch, right? First, and most importantly, you’re only going to want to use the, ‘Continuous Speech’ and ‘Restricted Continuous Speech’ options on spoken commands that are ***very distinct*** within your profile. How distinct they should be is totally up to you and your speaking style, so, you’re going to want to play around with that a bit.
Second, if you are using minimum confidence level thresholds, you’ll notice that the speech engine is a bit less confident while it’s in the middle of figuring out your continuous speech. You’ll find that you’ll probably be lowering your minimum confidence thresholds a lot for commands that do not have, ‘Normal’ set. I’d suggest not using anything but, ‘Normal’ on stuff like, ‘eject’, ‘eject cargo’, ‘self destruct’ and stuff like that ;) You’ve been warned (lol).

‘Advanced’ Button

Clicking this button will display the, ‘**Advanced/Experimental Command Options**’ screen. This screen displays some extended options of the command that are for advanced use or are experimental.

‘**Resource balance offset’ -** VoiceAttack throttles its resource usage to a certain degree. This is mostly to be a good neighbor to other applications as well as keep itself reasonably responsive. Every now and then, you'll run into instances where a little more speed could actually make a difference in your command. This experimental feature allows you to adjust the amount of resources your command will use when it is executing. The higher the value is set (from 1 to 100), the faster the command will run. However, resource usage of your PC will increase as a result and VoiceAttack itself may be more likely to become unresponsive depending on what you do in your command (for instance, infinite loops with a high value are probably not a great fit for this feature – you’ve been warned lol). Use with caution - for those of you automatically executing commands on profile load, remember that holding down CTRL + Shift when VoiceAttack is launched will bring up the load options screen ;)

Note if you've got a command with a handful of actions that you're not going to see much difference (if any). On the other hand, if you have a lot of actions in a command or have a bit of looping going on, you'll probably notice a performance gain. This may affect the timing of your commands, so prepare accordingly. Again, it's an experimental feature - experiment with it ;)

‘**Do not execute this command if it is already running**’ – This option will prevent the command from being executed if another instance of the command is already in progress. Ever have a long-running command and you forget that it’s already running and you accidentally run it again? Thought so. This helps with that ;)

6 - Action Management

Move action Up button

Click this button to move a selected action to an earlier part of the sequence. This can also be achieved by holding down the control button and pressing the up arrow key.

Move action Down button

Click this button to move a selected action to a later part of the sequence. This can also be achieved by holding down the control button and pressing the down arrow key.

Edit an action button

Click this button to modify the selected action (works the same as double-clicking on the selected item).

Delete an action button

Click this button to remove the selected action from the sequence (works the same as hitting the Delete key).

Undo

Click this button to undo the last change you made to the command lists.

Redo

Click this button to redo a change that has been undone.

7 - Command Type

VoiceAttack supports full commands (this is what you will probably be using almost exclusively) as well as composite (prefix/suffix) commands. Prefixes and suffixes only execute when they are used together in a composite voice command (which means that they will not execute on their own). When they are executed together, the actions from the prefix are executed first, followed by the actions in the suffix. This is handy if you want to create a lot of similar commands, without copying and modifying over and over again.

As an example, let’s use the actions found in a racing game. Let's say that the races potentially have up to 100 drivers. If you wanted to create commands to eject any one of the drivers from a race, you would need to create 100 commands ('eject driver 1', 'eject driver 2', 'eject driver 3', etc.). Next, if you wanted to mute any one of those drivers you would need to create another 100 commands ('mute driver 1', 'mute driver 2', 'mute driver 3', etc.). For every action that involves drivers, you would need to create another 100 commands. To solve this with prefixes and suffixes, you would first create the suffixes for the 100 drivers (yeah... I know that's a lot). The suffix actions would be something like this (for driver 82):

Press 8
Release 8
Press 2

Release 2
Press Enter
Release Enter

Once you have your suffixes lined up, you can then create as many prefixes as you need to work with them. For, 'mute driver', you create a command and designate it as a prefix. Its actions would look like this:

Press m
Release m
Press u
Release u
Press t
Release t
Press e
Release e
Press Space
Release Space

When you say, 'mute driver 82', all the actions from, 'mute driver' will be executed first, followed by the actions in suffix '82'. You only have to create one prefix and it is automatically paired with all of the available suffixes.

Prefix/Suffix Group -
Indicate a group name for your prefix or suffix to have the pairing of the prefix and suffix to occur only within that group. For instance, you can have a group called, 'taunt' where all of the suffixes contain something funny to say when you mute or eject a driver:)

8 - Repeating
VoiceAttack can execute the actions of a command once, or as many times as you like. To get VoiceAttack to repeat the command actions indefinitely, select the option labeled, 'This command repeats continuously'. To repeat the actions a certain number of times, choose, 'This command repeats X times' and fill in the number of times to repeat in the box provided.

To get VoiceAttack to stop repeating, you have a couple of options. The most heavy-handed way is to issue a command to stop all processing (this can be invoked from the main screen by clicking the, 'Stop Commands' button, by pressing the stop command hotkey(s) (see Options page) or by issuing a voice command that stops command processing (see Other Stuff screen). Another option is to issue a voice command that calls the, 'Stop Another Command' action. This can be found on the, 'Other Stuff' screen. **Note**: Looping is also available from within commands as of version 1.5.9.

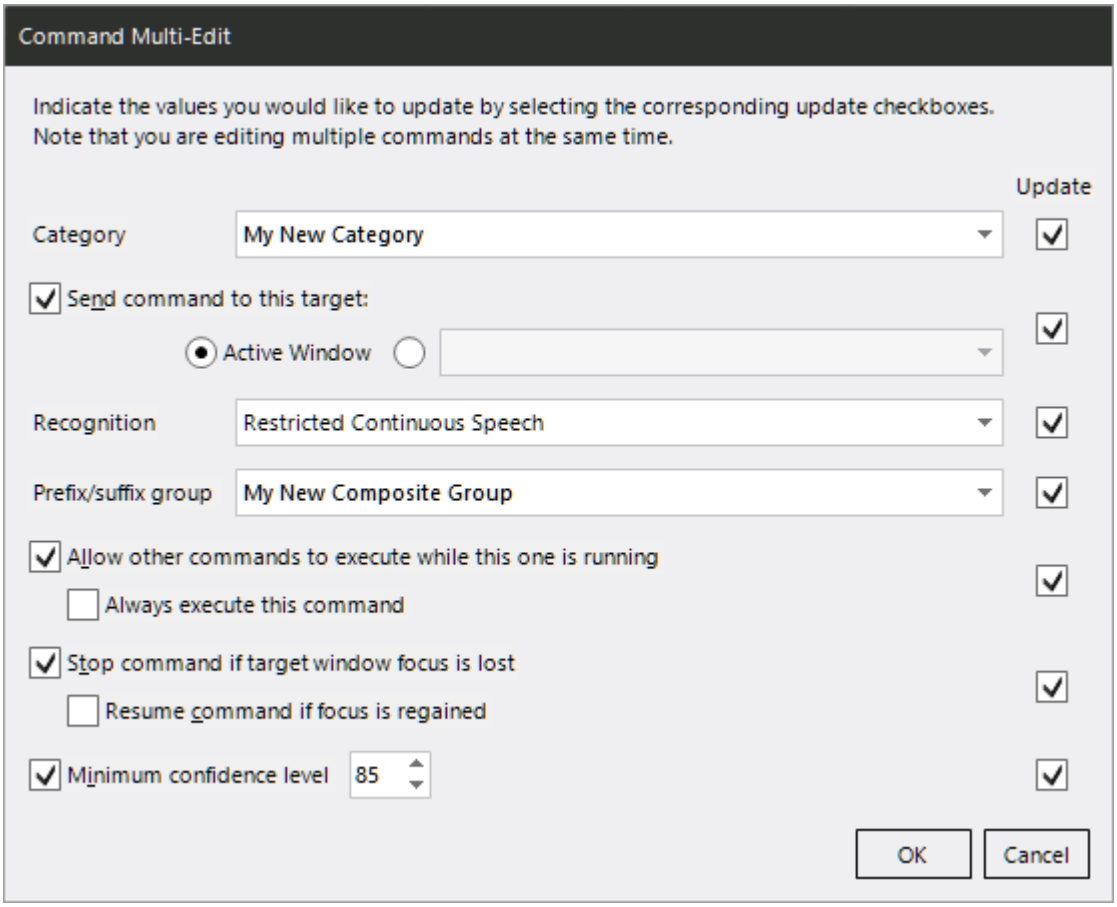
OK button
Click the OK button to commit all changes to the command.

Cancel button
All changes to this command will be undone if you click the Cancel button.

Note: Actions in this list can be copied and pasted within the same command action list as well as command action lists in other profiles. Most actions indicated above are also available in the right-click menu of the command action list.

Command Multi-Edit Screen

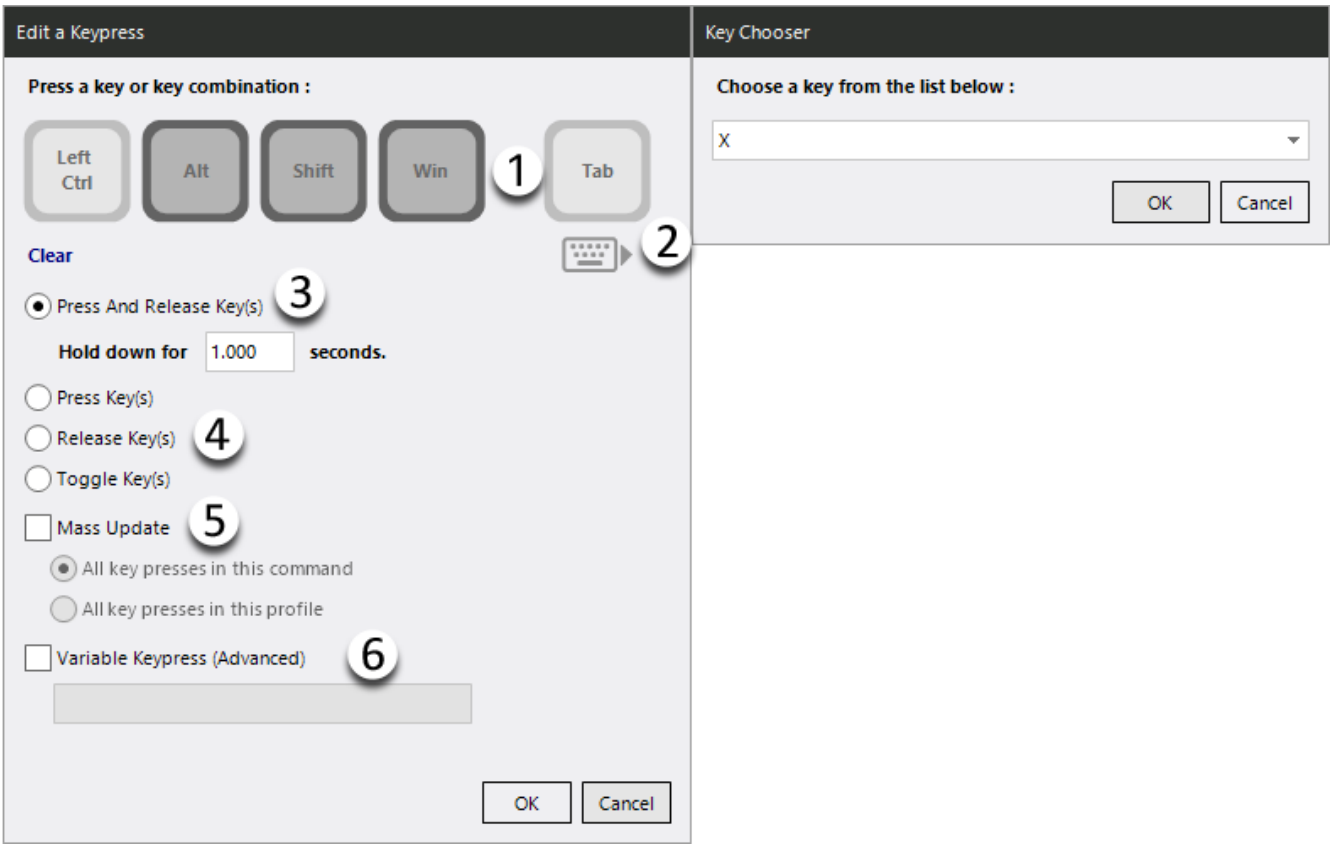
If you select more than one command at a time and click, ‘Edit’, you will be allowed to edit certain values all of the selected commands at once using the, ‘Command Multi-Edit’ screen.



This screen will allow you to choose which values will be updated for commands you had selected. Note that since blank and unselected are actually valid values, you will first need to select which values will be updated by checking the corresponding checkbox in the, ‘Update’ column. For instance, if you would like to update the Category value for your selected commands, you would first check the box to the right of, ‘Category’ and then fill in the input box. If you want all of the selected commands to have a blank value for Category, simply clear out the input box. Pretty straightforward ;) For more information on what each of the input fields represent within a command, see the, ‘Command Screen’ section earlier in this document.

Key Press Screen

This screen allows you to define a single key press for a command action (See 'Command Screen'). You are also allowed to specify a modifier with your key press. The modifiers that are available are 'Shift', 'Control', 'Alt' and 'Windows'. If your key press needs to be held down for a certain amount of time, you can indicate that time frame in seconds (up to a maximum of 99.999 seconds).



1 - Selected keys

This is where you indicate what keys to press. If you press, 'X', the right-most key icon will display, 'X'. If you press a modifier key (ctrl, alt, shift or Windows), one of the left-most key icons will display what you pressed. This will make up the key combination that VoiceAttack will send to your application.

2 - Key chooser

Clicking the mini keyboard pops up the, 'Extended Key Chooser' screen. From this screen, you can select any of the available keyboard keys (for example, if your physical keyboard does not have media or browser keys, you can select them from here).

3 - Press and release key(s) option

If you choose this option, VoiceAttack will press down and release a key. This option is further enhanced by using the, 'Hold down for X seconds' box. Enter a value here (in seconds) to indicate how long VoiceAttack is to hold down the key/key combination

before releasing. Note that a value of zero is **not** recommended for most games, as games tend to rely on polling for key state (a value of zero might make the key press occur too quickly for the game to react).

4 - Press key(s) option

Select this option if you only want VoiceAttack to press the selected keys down. This is usually used in a macro, with a subsequent, 'Release key(s)' action.

- Release key(s) option

Select this option if you only want VoiceAttack to release the selected keys. This is usually used in a macro, preceded by a, 'Press key(s)' action.

- Toggle key(s) option

Select this option if you want VoiceAttack to press a key if it is not pressed or release a key if it is pressed.

5 - Mass update

When you are **editing** a key press, you have the option to update all of the matching key presses in either the current command or the current profile to be the same as the one you are currently editing.
So, if you have a bunch of commands that press the, 'X' key and you want to change all of them to use the 'Y' key, simply choose one of your key press actions that presses the, 'X' key only and edit it. Change the key to, 'Y'. Then, choose mass update and then the 'all keypresses in this command' option. When you click, 'OK', all the actions in the command that had a key press of, 'X' will now be, 'Y' (all key press actions that did not have a key of, 'X' will be left alone).

Note that the keys and duration are the only attributes that are mass updated. The press method (down/up/press) are not updated.

Also note that hitting, ‘Cancel’ on the Key Press screen will not cancel a mass update. You will need to press, ‘Cancel’ on the Profile edit screen in order to cancel the operation, as the entire profile will be updated.

6 – Variable Keypress (Advanced)

Selecting this option will allow you to use a text variable to indicate the keys to press for this action (instead of using the icons at the top of the screen). This is to aid (primarily) in cases where keys for various commands may change periodically (such as with key bindings for games). The idea is that keypress variables would be initialized at profile startup or by plugin activation, thereby not requiring constant, manual updating of commands each time the key bindings change in a game.

Usage is pretty straightforward. To turn on variable keypresses for this action, make sure the box is checked and simply put the name of the text variable to use in the provided input box. The previously-set **text variable must contain properly-notated text in order to work**. The good news is that the notation is (almost) exactly the same as what you will find in Quick Input. So, for instance, if your command is to raise your landing gear and the keypress (for now) is ALT + L, set a text variable’s value to ‘[ALT]L’ (no quotes).

Note that the, ‘L’ does not have brackets. Keys with a single-character identifier (A-Z, +, ß, ç, etc.) do not need brackets. Special keys, such as Enter, Shift, Alt, F12, etc. will require brackets (see the section titled, ‘Quick Input and Variable Keypress Key Indicators’ for all the possible key indicators). Note also that there is no space between [ALT] and L. Spaces are actually picked up as key presses here, so if there is a space, the space bar will be pressed.

Put the name of the variable in the input box, and when the action is executed, the appropriate key method is performed (press, down, release, toggle) with the keys indicated in the variable. Note that with multiple keys that the order that the keys go down are the order that you provide. This happens virtually instantaneously, but order is still important. So, in this case, ‘[ALT]L’, the Alt key is manipulated first, and then L. **When releasing keys, the order is reversed.** In this case, the L key would be released first and then the Alt key. This is so you can use the same variable to press keys down and then release the keys in the proper order without having to create another variable.

Important: Since we are pressing keys and not (necessarily) generating characters (as with Quick Input), the keys that are pressed will be unmodified keys. So, for instance, if you are using an English keyboard and you put in, ‘@’ as the keypress you will notice that 2 key will be pressed. In order to reproduce the, ‘@’ as a keypress, you will need to modify the keypress yourself by including Shift, like so: [SHIFT]2 or [SHIFT]@ (either of these will work). This is the same as it has always been with using keypresses, however it seems a little more pronounced now.

Important: As an interesting side effect, you can pretty much manipulate as many keys as you want at once, including all of the modifier keys (LAlt, RAlt, LShift,RShift, etc.). Remember, again, they will all occur in sequence in the order you provide (no pauses between). Multiple instances of the same key repeated in a keypress may not have the desired result. ‘[ENTER][ENTER][ENTER][ENTER][ENTER]’ probably will not press the enter key five times, however, it will usually press it more than once due to timing (your system may vary). Just don’t do that lol.

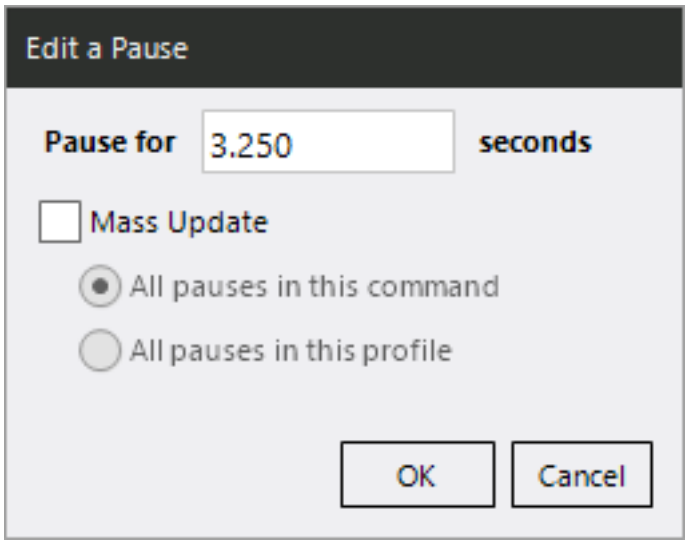
Note: The associated input box can also accept tokens (if, for some reason the variable name needs to be variable… yikes lol).

In the above example, at position 1, we want VoiceAttack to press 'Control' plus 'Alt' and 'X' keys all at the same time and release them after holding them down for 1 second. Note that 'Shift' and 'Win' are not selected.

When you click the 'OK' button, the indicated key press is added to your command action sequence. If the 'Cancel' button is pressed, you are returned to the Add/Edit Command screen, and **no changes are recorded**.

Pause Screen

This screen allows you to define a single, timed pause for a command action (See 'Command Screen'). This is useful for waiting between key presses or waiting for a program to launch. Simply add the amount of time you would like VoiceAttack to wait (up to 999.999 seconds) and click the 'OK' button. To cancel adding or editing a pause, click the, 'Cancel' button and no changes will be recorded.



In the example, above, we are adding a pause for 3 and a quarter seconds.

Note that in the, 'Edit a Pause' screen, you have the ability to mass update all the pauses in the current command or in the current profile. Just select the, 'Mass Update' option and then choose the scope of the change. This will only change pause actions where the value matches the original pause value. So, let's say we have a lot of pauses in our command that are 1 second long, but we want to change them all to 2 seconds. Just edit one of the pause actions that have 1 second, change the value to 2 seconds and then enable, 'mass update' and choose, 'all pauses in this command'. When you click, 'OK', all pause actions in the command that were 1 second will now be 2 seconds (pauses with different values other than 1 will not be altered).

Note that hitting, ‘Cancel’ on the Pause screen will not cancel a mass update. You will need to press, ‘Cancel’ on the Profile edit screen in order to cancel the operation, as the entire profile will be updated.

Variable Pause Screen

The Variable Pause screen works just like the Pause screen except instead of providing an exact pause time, you can specify a decimal variable to use, or any combination of tokens/literals that either resolve to a variable name or to a decimal value. The decimal variable must be set prior to using this feature (See, 'Set a Decimal Value'), otherwise the pause will not occur (since it will be zero seconds). Variable pauses can be used to have control over profile-wide pauses (change in one place, versus changing in many places). They can also be used with random values to give a slightly more natural feel to TTS, for instance.

‘Other Stuff’ Screens

Clicking the, '**Other**' button on the Add/Edit Command screen will display a fly out menu with several submenus: VoiceAttack Action, Sounds, Windows, Dictation and Advanced. Clicking on any of these submenus will display the various, 'special' actions you can add to your commands. Note that there is a little star up near the top. Clicking on this star will add this action type to your favorites which can then be accessed from the, 'Other' button on the Command screen. If you hold down CTRL while clicking the star, you can optionally clear all your favorites.

When you select a special action, you'll be presented with its corresponding add/edit screen.

Some items in here may be useful to you. Other items you will probably never use. There's a lot going on here, so, I'll try to do my best to explain each item. The special actions are organized below in the same manner they are organized in the submenus:

VoiceAttack Action

'Make VoiceAttack Start Listening'

Select this item if you want VoiceAttack to start listening. This is useful either by itself or in a macro. Note that there is a button the Main screen as well as a hotkey for this same action (See 'Options Screen' and 'VoiceAttack's Main Screen').
Also, note that if a, 'Start Listening' action is found AS THE FIRST ACTION within a command, the entire sequence will be executed as if VoiceAttack is already, 'listening'.

'Make VoiceAttack Stop Listening'

Same as above, except this stops VoiceAttack from listening.

'Toggle VoiceAttack Listening'

This starts VoiceAttack’s listening if it is turned off, and stops VoiceAttack’s listening if it is turned on.

'Make VoiceAttack Stop Processing All Commands'

Use this to stop all currently-processing commands. This works the same way as clicking the 'Stop Commands' button on the Main Screen. Note that this command action will only execute if your currently-executing command allows for other commands to run at the same time (see, 'Command Screen' - 'This command allows other commands to run at the same time' option). Note this will also stop any playing sounds or text-to-speech, and any keys that are pressed down will be released.

'Ignore an Unrecognized Word or Phrase'

This is tricky to explain. This basically just discards the recognized command and does not report it in the recognition log. I added this feature because sometimes the speech recognition engine picks up background noise and breathing as commands. You will see entries in the recognition log like: "Unrecognized Command: 'if if if'". Chances are, you will see things like this, too. Just add the irritating phrase as a command and select this option all by itself in the action sequence.

'Switch to Another Profile'

This will allow you to issue a command to switch over to another profile without having to jump out of your application to do so. If you select the option, 'Switch Profile by Selection', you will be presented with a list of available profiles in which you can switch. If you choose the option, 'Switch by name', you can freely type the name of your profile in the input box. Note that when using this option, if your profile names change, this value will not be updated. Also note that the input box can contain any combination of tokens that can be rendered to form a valid profile name (this is an advanced feature for those of you that do not know the names of the target profile until run time).

'Execute Another Command'

Selecting this will allow you to add a previously-created command to your command action list. Having nested commands keeps you from having to recreate entire action lists for duplicated functionality. Also, if any referenced command changes, the changes will be reflected in the nested commands.

There are two options for executing other commands. The first option allows you to select an existing command from a list. This is the safest way to execute other commands, since VoiceAttack knows ahead of time about any loops that may be encountered.

The second option (designated as an, 'advanced' feature) lets you select a command to execute by name (replacement tokens are supported). If a name of a command that exists is given, a simple loop check is done to make sure you will not potentially freeze up or crash VoiceAttack. If the referenced command does not exist (or if a replacement token is used) VoiceAttack will be unable to make the loop check, leaving you at risk for an infinite loop. Use this feature at your own peril :)
Note – If you are trying to execute a multipart/dynamic command by name, you must pick just one of the spoken commands that will be used. For instance, if you have a multipart/dynamic command with a 'When I say' value of, 'test;test [all;something]', do NOT use 'test;test [all;something]' in the box, as this will not work. You must put one of the derived spoken commands, such as, 'test', 'test all', or 'test something' in the box. Hope that makes sense ;)

The, '**Wait until this command completes before continuing**' option allows you to indicate whether or not subsequent actions will execute while the called command is running. If the box is checked, any commands that come after the executed command will have to wait until all the actions in the called command are finished processing.

Selecting the, '**Do not execute this command if it is already running**' option will prevent the chosen command from being executed if the chosen command is already executing. Ever have a long-running command and you forget that it's already running and you accidentally run it again? Thought so. This helps with that ;)

The '**Passed Values (Advanced)**' feature set of the 'Execute Another Command' action will allow you to pass various values to the command you are executing. In programmer-speak, it's most akin to passing in parameters to a function. The executed command will convert any passed values to command-scoped variables that

are accessible only within the executed command. This saves you the trouble of having to create and set variables prior to executing the subcommand. It also abstracts the variables used within the subcommand, so that the calling commands do not have to use same-named variables for the subcommand to read.

To maximize flexibility, the passed values must be included in a semicolon-delimited list. This list can contain any combination of literal values, variable names and tokens that resolve to each data type (note that the Decimals and Dates features currently only use variable names). The variables that are created within the executed command are named according to their data type and order that they are listed. For example, the first variable for a passed text value will be named, '~passedText1'. The second will be named '~passedText2' and so on. Note the tilde (~) at the beginning of the variable name. This indicates that the variable is command-scoped and is only available within the executed command.

Let's take a look at each type:

Text - This is used to pass text (strings) into the executed command. This can be any combination of literal values and text variable names separated by semicolons. Literal values MUST be enclosed in double quotes. Literal values can also contain any number of tokens. Variable names must NOT be enclosed in double quotes (otherwise they will be considered literal text lol). Plot twist: text variables in this context can contain tokens that will be rendered.

Here is an example of a literal value, a variable name and a literal value with a token all being passed in at once:

"this is some text**"**;myTextVariable;**"**some more text {TXT:myOtherTextVariable}**"**

Note that the literal text is in quotes first, then a semicolon, then a variable name that DOES NOT have quotes, then another semicolon, then a literal value combined with a token within quotes. When this list of semicolon-delimited items is passed on to the executed command, the executed command will create a variable named, '~passedText1' that will contain the value, 'this is some text'. The number is 1 since it is the first item in the list. For the, 'myTextVariable' variable, the executed command will create a variable named '~passedText2' which will contain the value from the 'myTextVariable' variable. Its number is 2 since it is the second item in the list. Note that if, 'myTextVariable' is Not set, '~passedText2' will also be, 'Not set'. Lastly, the literal value/token combo of 'some more text {TXT:myOtherTextVariable}' will be rendered and the result will be placed in '~passedText3'. It is numbered 3 as it is the third item in the list. You can pass as many values in the list that you need to, and a variable will be created and numbered accordingly.

Integers - For brevity, this all works the same as the 'Text' stuff above, except for just a few things. First, the literal values for your integers do not need double quotes. Second, the variable names passed in must be for variables that are of integer type. Third, any combination of tokens or tokens and literals that are rendered must resolve to something that can be converted to an integer. If the token or token literal combo cannot be resolved to an integer, the resulting variable in the executed command will

be Not set. Last, the created variables in the executed command will be named '~passedInteger1*..n*'. For example:
155;myIntegerVariable;{EXP:(5 + 5)} will produce '~passedInteger1' in the executed command that will be set to 155, '~passedInteger2' that will be set to the value contained in the integer variable named, 'myIntegerVariable' and '~passedInteger3' which will be set to the rendered value of the token (which would be 10). Note that although the {EXP} token is rendered as text, the value is converted to an integer before being placed in '~passedInteger3'.

Booleans - For brevity, this works like text and integers except for a few of things. First, although the literal values for your Booleans do not need double quotes, they must be the English words, 'True' or 'False' (although case does not matter). Second, the variable names passed in must be for variables that are of Boolean type. Third, any tokens that are rendered must resolve to the text value of 'True' or 'False' (again, case does not matter). If the token cannot be resolved to 'True' or 'False', the resulting variable in the executed command will be Not set. Last, the created variables in the executed command will be named '~passedBoolean1*..n*'. For example:
false;myBooleanVariable;{TXTRANDOM:true;false} will produce '~passedBoolean1' in the executed command that will be set to false, '~passedBoolean2' that will be set to the value contained in the Boolean variable named, 'myBooleanVariable' and '~passedBoolean3' which will be set to the rendered value of the token (which would have the word, 'true' or 'false' be randomly selected). Note that although the token is rendered as text, the value is converted to a Boolean before being placed in '~passedBoolean3'.

Decimals - Again for brevity, this works just like the previous types except for a few things. The first thing to note is that a literal decimal value does not require double quotes, however, a literal decimal value does require you to express it in the invariant culture. What that means is that due to the differences in the way decimal values are expressed from locale to locale, we need to express the decimal value in a common way that works for everybody (this is mostly so profiles can be shared). With the invariant culture, commas are ignored, and a point indicates the fractional part of the number. For example, 333.44 would be interpreted correctly, but 333,44 would not be rendered properly (it would be rendered as 33344). The other item to note is that the variables created by the executed command are named '~passedDecimal1*..n*'. For example: 111.22;myDecimalVar;{DEC:myOtherDecimalVar} will produce '~passedDecimal1' that contains the value of 111.22, '~passedDecimal2' that contains the value of the variable named, 'myDecimalVar' and '~passedDecimal3' which will be set to the rendered value of the token (if the token renders to a valid decimal value).

Dates - Just as with decimals, date literal values do not require double quotes and must be expressed in the invariant culture. This means that you must pass your literal value in formatted like this:

M/D/YYYY (use if you want to just indicate the date) or
M/D/YYYY h:m:s.ffff (use if you want to indicate both the date and time)

M = The month. 1 would be January, 12 would be December.

D = 1 would be the first of the month, 15 would be the 15th.
YYYY = four digit year. 2050 would be a valid year.

Note that there is a space between year (YYYY) and hour (h).

h = The hour of the day, expressed in a 24-hour clock. 23 would be 11pm.
m = The minute. 30 would be thirty minutes.
s = The second. 50 would be 50 seconds (seconds and milliseconds are optional).

Note that there is a dot between the second (s) and milliseconds (fff)

fff = The millisecond. 500 would be 500 milliseconds (seconds and milliseconds are optional).

A literal value of 5/1/2020 23:40:10.250 would yield a date of May 1, 2020 at 11:40pm (with an additional 10 and a quarter seconds).

Although there are some loose variations on this format (such as 01 being OK to use as the month and two digits being acceptable for the year (with certain rules)), for brevity we are just going to cover what's above.

The variable names that are created in the executed command are named '~passedDate1..*n*'. For example: 5/1/2020;myDateVar;{DATE:myOtherDateVar} will produce '~passedDate1' that contains a date/time value of May 1, 2020 (time will be midnight since no time part was supplied), '~passedDate2' that will contain the value of the variable named, 'myDateVar' and '~passedDate3' that will contain the rendered date/time value of the token (if the token renders to a valid date/time value).

'Command Queues - Enqueue Command'

This action will allow you to enqueue a command so that it will execute in a specified order with other enqueued commands. Commands that are enqueued first are executed first, and subsequent commands that are added to the queue are processed next after the commands added prior are finished. This is a rather advanced action, so, for more details about command execution queues, see the section labeled, '**Command Execution Queues Overview**' later in this document.

The, 'Queue Name' input will allow you to indicate the name of the command execution queue that you will be adding your command. If a queue by that name does not exist (that is, have a running instance), a new one will be established and that queue will remain available until VoiceAttack is closed. Note that you can simply type in a name into this box, or, you can select a queue name from the dropdown list. This input box will also accept any combination of tokens to establish a queue name. Also note that you can have as many command execution queues as your system will allow.

Just like the, 'Execute Another Command' action above, there are two options for enqueueing commands. The first option allows you to select an existing command from a list. This is the safest way to execute other commands, since VoiceAttack knows ahead of time about any loops that may be encountered.

The second option (designated as an, 'advanced' feature) lets you select a command to execute by name (replacement tokens are supported). If a name of a command that exists is given, a simple loop check is done to make sure you will not potentially freeze up or crash VoiceAttack. If the referenced command does not exist (or if a replacement token is used) VoiceAttack will be unable to make the loop check, leaving you at risk for an infinite loop. Use this feature at your own peril :)
Note – If you are trying to execute a multipart/dynamic command by name, you must pick one of the commands that will be used (for instance, if you have a multipart command labeled, 'test;test [all;something]', you can just put, 'test' in the box.

The, '**Do not add command to queue if already enqueued**' option will prevent the command from being added to the named queue if the command is already contained in that queue. Note that if the designated command is not added to the queue, the, 'Start the queue when this command is added' option (below) will be overridden (that is, the queue will not be started, as the command is not added).

The, '**Start the queue when this command is added**' option is a convenience feature that will allow you to start the execution of commands in the queue immediately after the current command is enqueued. This saves you the extra step of having to explicitly add a 'Start' queue action (see below).

The '**Passed Values (Advanced)**' feature set of the 'Enqueue Command' action will allow you to pass various values to the command you are enqueueing. If you are a programmer, this is most akin to passing in parameters to a function. The enqueued command will convert any passed values to command-scoped variables that are accessible only within the enqueued command. *** In order to save space (and to hopefully save some trees), this whole feature is explained just a short way up above in the, 'Execute Another Command' action section. Just search for, 'Passed Values (Advanced)' - it all works exactly the same for the 'Enqueue Command' action. ***
The **Evaluate passed values when command is queued** option lets you indicate when the passed values will have their variables and tokens evaluated. If you do not select this option, the passed values will be evaluated at the time the enqueued command is actually executed. If this option is selected, the passed values are evaluated the moment the command is enqueued. This is to aid in timing, as an enqueued command's execution can be deferred indefinitely. **Note:** Command-scoped and command-shared variables can only be evaluated if this option is selected (otherwise they will be out of scope when the enqueued command is executed).

'Command Queues - Queue Action'

This action will allow you to invoke the various functions of your command execution queues, such as starting, stopping and pausing. You can perform your queue action against a specified queue or all queues at once.

The, 'Queue' option will allow you to indicate a specific queue in which to control. Simply type in the name of the queue that you would like, or select its name from the dropdown list. Note that this input box will accept any number of tokens to resolve a queue name. Selecting the, 'All Queues' option will indicate that you would like the action to be performed against all queues. For instance, maybe you want to stop all of

the queues you have running all at the same time.

Next, you will want to choose the action to perform on your queue:

Start - This will make your queue start executing commands in the order that they were added. You can also start your queue by selecting the, 'Start queue when this command is added' option when you enqueue any command. The, '**Stop after all commands complete**' option indicates that after all the commands in the queue have executed that the queue should be stopped automatically. Tip - You can pre-fill a queue with commands and then start the queue at any time.

Pause - This will tell the queue to pause command execution once the currently-executing command has completed. Tip - You can add commands to the queue even when the queue is paused.

Unpause - This will get the queue executing commands again after it has been paused.

Toggle pause/unpause - This will pause or unpause, depending on the pause state (that is, it will unpause a paused queue, and pause an unpaused queue).

Stop - This stops all queue processing. This will halt the currently-executing command and then clear out any remaining commands that are contained within the queue. Note that a 'Stop all commands' action will also act as a Stop on all queues. Tip - You can add commands to the queue even when the queue is stopped.

Stop, but allow current command to complete - This will do everything the Stop action will do, except the queue will allow the currently-executing command to complete.

'Enable / Disable / Toggle Hotkeys'

Selecting these will allow you to turn on, off and toggle the keyboard hotkey shortcuts.

'Enable / Disable / Toggle Mouse Shortcuts'

Selecting these will allow you to turn on, off and toggle mouse button shortcuts.

'Enable / Disable / Toggle Joysticks'

Selecting these will allow you to turn on, off and toggle joystick button detection.

'Stop Another Command'

This option will let you specify a certain command that needs to be stopped. You will want to use this in conjunction with commands that loop or commands that have long-running macros. In earlier versions of VoiceAttack, the only way to stop running commands was to hit the, 'Stop Commands' button. Now you can indicate specific commands. **NOTE -** all instances of a command will be stopped, so, if you have multiple instances of a looping, asynchronous command, calling this will stop ALL instances.

'Quick Input'

This action will allow you to indicate text that you want typed out in your application. This differs from the recorder screen, as it allows you to include text tokens for replacement (see the section on tokens further down in this document). Simply type the text you want typed out in the, 'Text' box. You can then specify how long to hold down your keys by indicating a value in the, 'Hold down keys for X seconds' box, as well as specify how long to wait between keys by indicating a value in the, 'Wait for X seconds between keys' box. Note that a value of zero for either of these delays is not recommended for DirectX games, as they tend to rely on polling for key state). In order to represent keys such as, 'Enter', 'Tab', 'F1', etc., you can use some special indicators enclosed in square brackets: [ ]. For instance, if you want to press the 'Enter' key, simply include [ENTER] in your text. Some keys, such as 'Shift', 'Alt, 'Ctrl' and 'Windows' need to be held down while you type other characters. There are some reserved indicators for this as well. As an example, for the, 'Shift' key, [SHIFTDOWN] and [SHIFTUP] are provided. If you need to specify a pause between keys, you can use the [PAUSE:*seconds*] indicator, where *seconds* is the number of seconds to pause (Ex: [PAUSE:0.5] will pause one half a second, and [PAUSE:2.5] will pause for two and a half seconds).

Adding the following text to the Quick Input, 'Text' box:

Hello, out there![ENTER][ENTER]How are you?

Produces the following output:

Hello, out there!

How are you?

The full list of Quick Input key indicators is near the end of this document.

Note: Key indicators are not case-sensitive.

'Reset the Active Profile'

This action will reload the current profile. This is typically not something you'll want or even need to do for the most part, but is available for those that need it for some more advanced application (it's why the notes about this kinda wander off into the obscure o_O). When the profile is reloaded, any profile-scoped variables (variables prefixed with ONE '>') will be cleared. Persistent profile variables (variables prefixed with '>>') will be retained (see section below regarding variable scope if you're needing more info on that). All commands will be reloaded and any tokenized, 'when I say' phrases will be reevaluated. The speech engine will also be reloaded (of course), and any executing commands will be stopped. As you can see, this is whatever a, 'change profile' action does, just without actually changing ;)

'Refresh Variable Hotkeys'

This action works in conjunction with the, 'Use variable hotkeys' feature (see the, 'Command Screen' section for more information on setting up variable hotkeys). What this action does is refresh the hotkeys that VoiceAttack is monitoring based on the

current state of the variables that the variable hotkeys are using. For instance, let's say we have a command using a variable hot key, and the variable that is being used is, 'myTextVariable'. If, 'myTextVariable's value changes, VoiceAttack will not be aware of this change until you execute this action. Note that this is an advanced feature.

Sounds

'Say Something with Text-To-Speech'

This action will allow you to type in a phrase to be spoken by your built-in Text to Speech engine. Something fun you can do with this is input several phrases at once, separated by a semicolon and VoiceAttack will randomly pick a phrase to, 'say'. For example, you can input, 'Fire Weapons;Fire At Will;Destroy Them All' and VoiceAttack will see this as three random phrases for the same command.

If you need further dynamic responses, you can include them by putting the responses between square brackets (just like dynamic spoken command phrases from the Command screen). Putting text between square brackets in text to speech is called a, 'dynamic response section'.

Dynamic response sections allow you to specify a part of your text to speech (TTS) that may vary. Sometimes you may want TTS to say, 'Hello captain' and sometimes you may want it to say, 'Greetings, captain'. To indicate that you want to use a dynamic response section, enclose the section in square brackets: [ ], with each element separated by a semicolon. Your TTS phrase may look something like this:

[Hello;Greetings]captain

This will result in either, 'Hello captain' or, 'Greetings captain'.

Note that you can still add phrases separated with a semicolon:
[Greetings;Hello]captain;Hi

With this example, the result will be either 'Greetings captain', 'Hello captain' or 'Hi'.

The dynamic response sections don't have to just be at the beginning. They can be anywhere in the phrase. Also, as a side-effect, if you put a semicolon on at the end of the selections, it makes the section optional:

[Greetings;Hello]captain[how are you;]

This results in a response of:

Greetings captain how are you
Hello captain how are you
Greetings captain
Hello captain

Note that there is a semicolon after 'how are you' to indicate that the entire section is optional.

Something to consider when using this feature is that you can create a lot of permutations from very few words. Use with care :)

Dynamic response sections can also contain **numeric ranges.** To indicate a numeric range in a dynamic response section, just include the minimum and maximum values separated by an ellipsis (..). Not sure how many applications this may have, but it's there for you (it's available for dynamic commands… just thought I'd leave it in).
A bad example would be [Greetings;Hello]captain. I tried to call you [2..10] times today
This will include responses that look like this:

Hello captain. I tried to call you 5 times today
Greetings captain. I tried to call you 10 times today.

Note that you can preview and set the **volume** and voice **rate** of your phrase from this panel.

The, '**Voice**' drop down box will allow you to select the speaking voice that you would like to hear when your text is spoken. Note that you can also type freely into this box. What can be typed in the box are text variable names, literal text and/or any combination of tokens. Note that whenever whatever was typed in is resolved, it must match an installed voice name exactly, otherwise the default voice will be used.

Advanced: If, 'Default' is selected as the, 'Voice' value (or, if a token or variable name does not resolve to a valid voice name as indicated above), the text-to-speech voice selected in Windows' Control Panel will be used. You can override this voice by opening the 'Profile Options' screen and selecting a voice from the, ''Default Text-to-speech voice' option. See, 'Profile Options' screen for more details.

Speech Synthesis Markup Language (SSML) is supported if you want to do some more fancy stuff. Visit Microsoft's site for more information on SSML: http://bit.ly/1PisKMD.

Certain tokens can be used with the Text-To-Speech phrases. See the section way down at the end titled, 'Text-To-Speech Tokens'.

Another advanced item is the Text-To-Speech **Channel** feature. If you are using the, 'Integrated Components' audio output type option (Options screen, audio tab), you will be given the ability to route the Text-To-Speech audio out of a selected audio playback channel. Simply select the audio channel from the dropdown list where you would like the audio to be routed. Selecting, 'Default' for this feature will not route the audio, and Text-To-Speech audio will be rendered through the default audio playback device as specified in Control Panel.

If you would like to give your text-to-speech a little bit of extra flair, you can apply an '**Effect Set**' by selecting it from the list. You can also manage your effect sets from this screen by clicking on the, '…' button. See the section titled, 'Sound Effects' later in this

document for more info on how to set these up.

There are two options available for speech execution. Checking the, 'Wait until speech completes' option will hold up the executing command until the speech is finished. Checking, 'This completes all other text-to-speech' will stop any other speech that is currently running. Note that it says, 'complete' rather than, 'stop' or, 'interrupt'. Any commands that are currently waiting on speech to finish will immediately resume (as their pending speech actions would then be, 'completed').

The, ‘Mass update’ option is an advanced option that allows you to update all, ‘Say Something with Text-to-Speech’ actions in the current profile. Whatever voice that was originally selected will be updated to the newly-selected voice. So, if you change the voice from, ‘Default’ to ‘Microsoft Hazel’ and select the, ‘Mass update voices’ option, all actions in your profile that currently use the, ‘Default’ text-to-speech voice will be updated to, ‘Microsoft Hazel’. Also, any action that contains the originally-selected volume and/or rate, will be updated to reflect the currently selected volume and/or rate. So, if you change the volume of, ‘Default’ from 100 to 90 and select, ‘Mass Update’, any, ‘Say Something with Text-to-speech’ actions that are using the, ‘Default’ voice and have a volume of 100 will be set to 90. All others will be ignored.

Also note that hitting, ‘Cancel’ on this screen will not cancel a mass update. You will need to press, ‘Cancel’ on the Profile edit screen in order to cancel the operation, as the entire profile will be updated.

'Play a Sound'

This feature simply plays a sound file that you choose. VoiceAttack has three audio output types that you can select from in order to play your sounds. The audio output type can be selected from the Options screen on the, ‘Audio’ tab (see, ‘Audio Output’ in the Options screen section later in this document for descriptions of each type). The reason this is important is because there are certain rules and various options to consider that are available for each output type.

When selecting, ‘**Legacy Audio**’ as your audio output type, you can **only** play .wav files. That’s about it. You **can’t** set the volume, balance or channel, plus you will not be able to have VoiceAttack wait for the sound to complete (legacy audio is always played asynchronously in the executing command). Why is this even available? Well, it’s there in case you need it. This was the first method that VoiceAttack employed for playing audio way back at the beginning and was left in just in case the other output types just will not work for you (doesn’t hurt to leave it in, right?).

If, ‘**Windows Media Components**’ is selected as the audio output type, you have a wide variety of file types that you can play. You can also adjust the volume or set the start and end positions. This was the second audio output type added to VoiceAttack, and was left in not only to ensure backward-compatibility, but also because the components work extremely well with files that just won’t seem to play otherwise.

When, ‘**Integrated Components’** is selected as the audio output type, you will have access to all of the available, ‘Play a Sound’ options, as well as a wide variety of file

types to play. The only drawback is that this mode may be a little bit pickier about the audio files that it plays, so make sure you preview your sound to make sure it's going to work for you.

The, 'Play a Sound' feature works much like the 'Run an application' feature above. Click the file browser ('…') button to select a sound file (note again that legacy audio mode is restricted to .wav files, and that 'Integrated Components' or 'Windows Media Components' allow you to play .wav, .wma and .mp3 (also .ogg, .flac, .m4a and .aac if you have the proper codecs installed) files). Note that this input box will also accept any combination of replacement tokens (See, the 'Text (and Text-to-Speech) Tokens' later in this document for more details). Note that the input box is also a dropdown list. If you would like to play one of VoiceAttack's internal sounds and, 'Integrated Components' is selected as the audio output type, simply drop down the list to select one of these sounds. You'll also notice that the path to the internal sounds is always, 'internal:', followed by the sound name (for instance, 'internal:Abinkle' (without the quotes) will indicate the internal sound, 'Abinkle').

You can preview your sound file by clicking the, '**Preview**' button.

The options available for both 'Integrated Components' and 'Windows Media Components' audio output types include being able to select the volume of the sound you are playing. You also have several additional options. The first is, **'Wait until sound completes before continuing'**. This will hold up the containing command until the audio finishes. The next option is, '**This completes all other sounds**'. This will stop any other sound that is currently playing. Note that it says, 'complete' rather than, 'stop' or, 'interrupt'. Any commands that are currently waiting on sounds to finish will immediately resume (as their pending sound actions would then be, 'completed'). The third option is the, '**Begin at position**' option. This will allow you to start the sound playback at a certain number of seconds, expressed as a decimal value. Note that this box can accept a decimal variable name as well as a token that resolves to a decimal value. See also the, '{STATE_AUDIOCOUNT}' and, '{STATE_AUDIOPOSITION}' tokens later in this document. Also note that, 'Begin at position' is not available when using Legacy Audio as your selected output type. If you choose to start your audio at a certain position, the, 'Fade in' option will become available. Fading in may help make the audio sound better when started in positions that are loud. Just a very minor enhancement… no big deal ;) The fourth option is, '**End at position**'. This works exactly like the, 'Begin at position' option above, except this option marks where you want the audio to end within the playing sound (also, you have a, 'fade out' option instead of, 'fade in').

The options available only to the, 'Integrated Components' audio output type are Balance and Channel. **Balance** allows you to adjust the playback audio to your left or right speaker. For example, sliding to the left will increase the volume in the left speaker, and decrease the volume in the right. **Channel** allows you to choose the device on which your audio will be played. So, if you would like a certain sound only played back through your desktop speakers, you can choose to do that here. Choosing, 'Default' will play the audio back through the default playback device specified by Windows.

The, '**Variable Volume**' feature of this screen is available as an advanced option to allow you to specify a variable that sets the volume for the played sound. If the value in this box can be resolved to an integer value, that value will be used to override what is set by using the volume slider higher up on the screen. The value must be between 0 and 100, with zero being no sound and 100 being full volume. The input box can take a literal value, a variable name or any combination of tokens that will resolve to a valid value. Note that since this option is command-dependent, 'Preview' will not resolve this value from this screen.

The, '**Variable Balance**' feature of this screen is available as another advanced option to allow you to specify a variable to set the balance for the played sound. If the value in this box can be resolved to an integer value, that value will be used to override what is set by using the balance slider higher up on the screen. The value must be between -100 and 100, with -100 being full left, 100 being full right and zero being center (full left and right). The input box can take a literal value, a variable name or any combination of tokens that will resolve to a valid value. Note that since this option is command-dependent, 'Preview' will not resolve this value from this screen.

Prefetch (available only if, 'Integrated Components' is selected as the audio output type in Options) - This advanced option allows you to indicate to VoiceAttack that the audio file associated with this action is to be loaded into memory when the profile is loaded. The purpose of this is to decrease the amount of time it takes for an important audio file to start playing (instead of taking time to read the data from disk at the moment that it is needed).
Note: If you are using a profile load command (that is, you have, 'Execute a command each time this profile is loaded' indicated in Profile Options), the audio prefetch will occur immediately following this command's execution. This will allow you to initialize variables for file names if necessary.

Also, the reason this option is indicated as an advanced feature is because **there currently is no cap on the amount of memory used to store prefetched audio**. That means you can prefetch so much audio that you'll run VoiceAttack out of memory – and that will cause a crash (VoiceAttack attempts to prevent a crash by dumping the prefetch, but depending on the load, that may not be enough to avoid a crash). My advice is to use this option sparingly. For additional prefetch options, see, 'Disable Audio Prefetch' and 'Preserve Prefetch Audio on Profile Change' options as part of the Options screen later in this document.

Tip: If you are unable to hear your sound file, you can always go into the Options screen and try out a different audio output type from the, 'Audio' tab.

Advanced/Beta - You can access sound resources in resource libraries that you create. This is an advanced feature that most will never even need, but it's there if you need to use it. Come on by the VoiceAttack user forum's, 'How Do I?' section (https://forum.voiceattack.com/smf/index.php?topic=2962) for more information on how to set this up.

'Play a Random Sound'

This action will allow you to make a selection of sounds that will play randomly. You can choose to select individual files or entire directories of files.

To add individual sound files to the list, click on the radio button titled, 'Play a random sound file from a list' then click the, 'Add New' button. You will then be presented with the same interface as found in the, 'Play a Sound' action (see above). For every sound file selected, you will be able to choose its volume and whether or not it holds up the macro until it completes or stops other sounds from playing. To edit a sound, just click on the, 'Edit' button (or double-click the item in the list). To remove a sound, click the, 'Remove' button. Note that you can select multiple files at once for adding, editing or removing.

To play a random sound out of a directory of sounds, choose the option labeled, 'Play a random sound from a directory. You will then be able to choose the desired directory, plus set the options for all the files that will be played (note that the options apply to all files in the directory. If your files need to have their own attributes, you will need to use the first option above). All supported files (.wav, .mp3, .ogg, .flc, .m4a, .aac) will be used out of the selected directory as well as shortcut files (.lnk) that are for sound files.

As with the, 'Play a Sound' action up above, you can choose the audio channel that your randomly-played sound will be rendered. Just select the desired output channel from the dropdown list. See the, 'Play a Sound' action for more info on selecting an audio channel.

The, '**Suppress Repeat**' option will help minimize the chance of constantly repeating sounds.

'**Prefetch**' works just like it does with the, 'Play a Sound' action above (by preloading audio files) - including all the bad things that can happen. The difference here is that entire directories or huge lists of audio files can be selected at once, **furthering the need to be extra cautious**. Just use good sense with this one and you'll be alright ;)

'Captured Audio'

For mostly fun, or to possibly diagnose a mic problem, there is the, 'Captured Audio' feature. What this does is play back or save the audio that VoiceAttack receives as input. There are several types of audio you can play back (selected from the, 'Type' list):
Recognized Audio – This is the current recognized audio. So, if you say, 'fire weapons' and this action is in your command, you will hear your very own voice say, 'fire weapons'. Yup… you sound that bad, for reals.
Previous Recognized Audio – This is the recognized audio prior to the current recognized audio. This is so you can issue a voice command and hear the horrible thing you just said (and not the current horrible thing you just said).
Unrecognized Audio – This is input that was considered by VoiceAttack to be, 'Unrecognized'. For me this seems to always be a lot of seemingly aggressive typing sounds.
Captured Audio – This is the latest non-dictation input that is recognized or

unrecognized.
Previous Captured Audio – This is the second-to-latest audio input. Again, just in case you want to get this audio using a voice command.
Dictation Audio – This is the audio captured when dictation mode is enabled.

There are some options available when playing back the captured audio when using, 'Integrated Components' (Options screen). You are able to choose the volume and output channel, as well as be able to have the command wait until the audio is finished. When the audio is played, you have to option to complete (stop) all other audio as well. The last option for this feature is to be able to save the captured audio to a file. Select the, 'Save the captured audio' option and then indicate the path where you would like the file saved. Note that this box accepts any combination of text and tokens. This feature will NOT overwrite a file that exists. If you want to save with unique file names, make sure to check out the {TIMESTAMP} and {GUID} tokens later in this document. Note also that the file that is saved is in .wav format only.

Windows

'Run an Application'

Select this option if you want VoiceAttack to launch a program. There are a few extra input boxes with this action:

Path of the program to be run

This is what program will be launched by VoiceAttack. You can click on the file browser button (labeled, '...') to browse for a file.

Note: You can drag and drop files to this input box (edit: maybe... depends on the security of your system).

Note: I won't go into much detail, but, you can also execute shell commands with Windows Vista and later (type in a value like, 'shell:MyComputerFolder'). Check out the web for more details on shell commands (very handy).

File browser ('...') button

Use this button to select a file to launch. This doesn't have to be just executable (.exe) files for games and apps. This can also be files that Windows have associated with other applications (for example, launching, 'helloWorld.txt' would launch the Windows Notepad.exe program and load the, 'helloWorld.txt' file. Kinda handy....

'With these parameters' box

If your launched program needs extra run-time parameters, you can specify them here. The example shows us launching Notepad with 'c:\mytext.txt' as the parameter. When this action is activated, VoiceAttack launches Notepad and displays the contents of, 'mytext.txt'.

'In this working directory' box

Specify the working directory for your launched application here.

'Window Style' selection

Indicate how your launched application will behave when it is launched. Note: that the target application must support the attribute specified.

Advanced
This section is for advanced use of, 'Run an Application'.

'Do not wait for launched application' option - This option allows the command to continue on without waiting for the launched app (this is the default option).

'Wait until launched application is started before continuing' option - This option tells VoiceAttack to wait until the launched application is started and is ready to receive input before continuing to the next action. You can specify how long to wait before giving up by selecting the, 'Only wait up to *X* seconds' option and filling in the box (see below). If the application does not launch, or, the time indicated is exceeded, you can exit the command completely if you select the, 'Exit command if launch has failed or wait time exceeded' option. This option also has its own option called, '**Set command to target launched application'**. This option will make the launched application the process target for the remaining duration of the command and all subsequent input operations will be directed to it (this overrides the currently selected process target). See the guide titled, '**Application Focus (Process Target) Guide**' later in this document.

'Wait until the launched application exits before continuing' option - This option will keep the command from continuing to the next action until the launched application closes. There are two options that go along with this feature: 'Capture STDOUT to a text variable' and 'Capture Exit Code to an integer variable'.
'Capture STDOUT to a text variable' allows you to specify a text variable to hold the STDOUT generated by the launched app. The text variable can then be used as you wish (for instance, in a condition block or as a plugin parameter). This option is enabled by simply indicating the variable. Leave this blank to leave this option turned off.
'Capture Exit Code to an integer variable' works exactly like the STDOUT feature, except you are capturing the app's exit code an integer variable. Again, leave this field blank to leave this option turned off.
You can specify how long to wait before giving up by selecting the, 'Only wait up to *X* seconds' option and filling in the box (see below). If the application fails to launch, or, the time indicated is exceeded, you can exit the command completely if you select the, 'Exit command if launch has failed or wait time exceeded' option.

'Only wait up to X seconds for launched application' option – This option is available for both of the wait features listed above. Specify how many seconds to wait for the application to either launch or exit. If the time is exceeded, the command will continue on to the next step.

'Exit command if launch has failed or wait time exceeded' option – This option will cause the command to exit if the application fails to launch or exceeds the time indicated in the, 'Only wait up to X seconds…' option.

'Stop a Process by Name'
Select this if you want to terminate a running process. The drop down list shows all running processes, as indicated by Windows. If you know the name of the process

that you might like to terminate, you can select it here. Note that you can type into this box and that tokens can also be used. Use this functionality with great care.

'Set a text value to the Windows clipboard'

This action will allow you to put a text value in the Windows clipboard. The value can also contain text tokens or condition tokens.

To clear the clipboard, leave the value in the box blank.

The value that is stored in the clipboard can be accessed with the, '{CLIP}' token (see section on Tokens near the end of this document).

Note: Pasting from the clipboard will require a command that executes a CTRL + V action (or whatever your system supports).

'Perform a Window Function'

This action will let you target a particular window/process (main window of a process) and perform a particular action on it. To select a window of an application that is currently running, just drop down the box labeled, **'Window Title'**. You can also choose the currently active, focused window by selecting, '[Active Window]' from the list. Note that this is a free input box and you can modify your selection (as detailed below). This box also accepts text variable names as well as any combination of tokens if needed.

The value in the drop-down box can contain wildcards indicated by asterisks (*). This is handy when the title of the window changes. To indicate that the window title **contains** the value in the box, put an asterisk on each end. For instance, if you want to target any window that contains, 'Notepad' in the title, put, '*Notepad*' (without quotes) in the box. To indicate that the window title **starts with** the value in the box, put an asterisk at the end: 'Notepad*'. To indicate that the window title **ends with** the value in the box, put an asterisk at the beginning: '*Notepad'. The values are not case-sensitive. The first window found to match the criteria indicated will be selected. **Advanced**: Note that you can also use process names as they appear in the Windows Task Manager. You can use wildcards the same as you do with the window titles. Window titles are checked first, and then the process names. **Really advanced**: A numeric process id can be searched for if the value in the box (which can be a text variable/token) resolves to an integer value. Also, if you find that you need to search by window class name, you can use this input box to search for it as well. In order to search for a window by its class name, simply prefix the search term with a, '+' (plus sign). For instance, if you are searching for a window that has a class name of, 'foo', simply put, '+foo' in the input box. If needed, you can select, '[Active Window]' from the dropdown list (instead of using tokens the old-fashioned way ;)). Note also that you can click on the, 'Title', 'Process' or 'Class Name' links near the bottom to fill in this box (see, 'Active Window Details' section below). Note that if you choose, 'Class Name', the, '+' is prefixed for you.

VoiceAttack will also allow the action to pause for a specified amount of time while trying to acquire the window and make sure it is receiving messages. If the pause

expires, nothing will happen. If the window is found, processing will continue immediately. To set the maximum amount of pause time, just check the box labeled, '**Pause up to**' and set the number of seconds (or fractions of seconds) to the desired value.

If the wait time is exceeded, or, if the window/process are not available, you can set the option, '**Exit command immediately if window/process not available**' to jump out of the command instead of allowing the command to continue processing.

Once we have a handle on the window, there are several things that can be done to it:

'Display' - This will let you show/minimize/maximize/hide/etc. your targeted window. Just drop down the list and choose one of the following (**Note:** Yeah, this seems a little confusing. You will probably need to try out various methods to see which one works for you. The list could have been pared down to just a few simple items (show/minimize/maximize), but the underlying feature allows for more flexibility. In order to provide this flexibility, all the functions have been exposed to the user, both in name and description. So, basically, these will work differently based on situation, and, quite frankly, I'm not entirely sure if they are all even necessary for most (just left in for your use, if you can use 'em)) :

Normal - Activates and displays a window. If the window is minimized or maximized, the system restores it to its original size and position.

Minimize - Minimizes the specified window and activates the next top-level window in the Z order.

Maximize - Maximizes the specified window.

Show Minimized - Activates the window and displays it as a minimized window.

Show Maximized - Activates the window and displays it as a maximized window.

Show Minimized No Activate - Displays the window as a minimized window. This value is similar to 'Show Minimized', except the window is not activated.

Force Minimized - Minimizes a window, even if the thread that owns the window is not responding.

Normal No Activate - Displays a window in its most recent size and position. This value is similar to 'Normal', except the window is not activated.

Show - Activates the window and displays it in its current size and position.

Show No Activate - Displays the window in its current size and position. This value is similar to 'Show', except the window is not activated.

Restore - Activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. You should specify this flag

when restoring a minimized window.

Show Default - Sets the show state based on the state of the window when it was created.

Hide - Hides the window and activates another window. Use with caution.

Set Always on Top - This will attempt to set the target window's state as always on top (that is, the top-most, focused window that remains on top of all other windows). Note that some windows within certain applications will not react to this state setting.

Unset Always on Top - This will attempt to remove the target window's 'always on top' state. Note that some windows within certain applications will not react to this state setting.

The 'Display' feature also has an additional option, **'Set command to target this window'**. This option will make the displayed window the process target for the rest of the command. This will override what you have set as the process target at the command, profile or global level. This is useful if you need to send input to a different application while you are in the middle of a command. There's a lot of extra detail in the section titled, '**Application Focus (Process Target) Guide**' later in this document.

'Close Window' - Closes the targeted window (surprise!).

'Move Window' - This will allow you to move the targeted window to a specific X, Y coordinate on your screen. Yup... you can move it right off the screen, so be careful. Note that you can click on the, 'Location' link near the bottom to fill in these boxes (see, 'Active Window Details' section below).

'Resize Window' - This will allow you to resize the targeted window to whatever width and height you like. Note that you can click on the, 'Size' link near the bottom to fill in these boxes (see, 'Active Window Details' section below).

'Change Title' - This will allow you to change the title text of the target window. If you find yourself with multiple instances of the same application open, this can be handy when it comes to targeting each instance. Note that this box respects replacement tokens.

'Change Opacity' - This will attempt to change the opacity (transparency level) of the target window, expressed as a percentage. A value of 100 would change the opacity to 100%, which means the target window will be fully visible without any transparency. A value of 0 would set the target window to fully transparent (use with caution lol). Note that some windows within certain applications will not react to this state setting.

The, '**Active Window Details**' helper section near the bottom will display information about the active window that you currently have selected, if you have selected a window outside of VoiceAttack itself. This section indicates the title of the active window as well as its process name, size and location. Clicking on the, 'Title' or, 'Process' links will copy the title text or process text value into the, 'Window Title'

dropdown box up near the top. Clicking on the, ‘Location’ link will copy the location of the active window ((X, Y) coordinates) into the, ‘Move Window’ boxes. Clicking on the, ‘Size’ link will copy the size of the active window (width, height) into the, ‘Resize Window’ boxes.

'Change Default Audio Devices'

This action will let you change Windows’ default multimedia and communications playback and/or default recording devices.

VoiceAttack uses Windows Media components and the Windows speech engine to do its thing. Windows Media components will **only** work with the default playback device, and the Windows speech engine can be set (and is set by default) to the default recording device. Sometimes you may want to change these devices when you are using VoiceAttack. For instance, you may want to switch over from a webcam mic / speaker setup over to a headset. This feature may save you a step (or several) while using VoiceAttack. To use this feature, simply select the devices you would like to use by checking the appropriate boxes and selecting the devices you would to use from the lists (the list only shows your currently-available devices). Now for lots of NOTES:

Note: If the Windows speech engine is set up to be something different than the default recording device (set up through the control panel), changing the default audio device will have no effect on the Windows speech engine, which means it will have no effect on VoiceAttack.

Note: This feature is only available for Windows 7 and later.

Note: If the device’s underlying identifier is changed (driver update, Windows update, crash, etc.) and this action is run, VoiceAttack will attempt to resolve the device by its last-known device name. If you see a log message containing, ‘Resolved by device name’, VoiceAttack has successfully made the change, but you may want to update this action as it is not automatically saved.

Note: There are other ways to change these devices. One way is through VoiceAttack startup in the Options screen. Another way is through command-line variables such as -input, -output, etc (see the section on command-line variables later in this document). There are also corresponding tokens ‘{STATE_DEFAULTPLAYBACK}’ and {STATE_DEFAULTRECORDING}’ (see the token reference later in this document).

<u>WARNING</u>: This is not a VoiceAttack setting, rather a Windows device setting and can (and probably will) cause other applications that depend on the changed devices to appear to malfunction. It’s not THAT big of a deal, but it will definitely throw you off when your Skype or TeamSpeak is not working how you left them.

'Windows Miscellaneous Functions'

Given proper version and user rights, VoiceAttack will attempt to execute any of the following Windows functions (they’re all mostly self-explanatory and/or probably don’t

need explanation):

'Toggle Desktop' - Toggles the desktop view in case the boss is coming. You'll need to come up with a creative command name here so he/she will think you are actually doing work.

'Lock Workstation' - Locks your PC so your roommate can't mess with your settings. Also, works to hide what you are doing from your boss ;)

'Log Off Current User' - Logs you out when you are done for the day.

'Sleep (Standby) PC' - Put your PC to sleep to save some power while you go and do stuff.

'Force Sleep (Standby) PC' - This will force your PC into sleep mode. What this does is it tells Windows to not alert any apps that it's going into standby mode. Take that, apps!

'Hibernate PC' - Hibernate your PC and save even more power.

'Force Hibernate PC' - Again, this tells Windows to dis all of the apps by not telling them that it's going into hibernation.

'Restart PC' - This will reboot your PC. This generally gives you a chance to save any unsaved documents

'Force Restart PC' - This will not only reboot your PC, but it will **NOT** give you a chance to save any unsaved documents. Use with care.

'Shutdown PC' - Shuts down your PC and will probably give you the chance to save your unsaved documents.

'Force Shutdown PC' - Just like the restart option above, only the PC just stays off. You will **LOSE** any unsaved work. Use with care.

'Power Off PC' - Works just like shutdown, except, where supported, will act like you pulled the plug out of the wall. This may have adverse effects on your system.

'Force Power Off PC' - Forces the power off of your pc. Just like all the other, 'force' options, you **WILL LOSE** unsaved data.

'Minimize All' - Minimizes all open windows.

'Undo Minimize All' - An undo for the, 'minimize all' you just did.

'Open Run Dialog' - Opens the run dialog so you can look like you know what you are doing… maybe even be productive.

'Open Search Dialog' - Opens the Windows search dialog so you can find those cat

videos easily.

'Open Window Switcher' - This will arrange all open windows in a tiled format for you to choose from.

'Run Screen saver' - Simply runs your screen saver if you have one that is active. “Why did you just shove this in here?” It was just one more option, might as well include it, right? Easier than adding it later. Coming soon: Espresso Maker

‘Hide Task Bar’, ‘Show Task Bar’ - Hides and shows the Windows task bar.

‘Reset Keyboard Shortcuts’ - Other software may override VoiceAttack’s keyboard shortcuts and listening hotkeys and cause VoiceAttack’s shortcuts and hotkeys to stop responding. Use this action to attempt to reset VoiceAttack’s shortcut/hotkey functionality. Note that when this action is executed, VoiceAttack may then be the application overriding other software ;)

'Set Audio Level'

This screen will allow you to add an action that will change a specified audio endpoint’s volume. This is useful if you want to voice control or hotkey various volume controls. You can choose to set the overall system volume, the volume of specific recording or playback devices (microphones, speakers, headphones) or the volume of various applications as they relate to the System Volume Mixer and the overall system volume (you can find the user interface of the System Volume Mixer in the system tray, or by right-clicking on the VoiceAttack icon in the task bar and selecting, ‘System Volume Mixer’).

To change the overall audio volume of your computer (most common), select, ‘System’. To change the volume level of a specific playback device (like speakers or headphones), select the, ‘Playback Device’ option and select the device you would like to modify. Choosing the, ‘Default’ device will always select the default playback device as indicated by Windows. Note that selecting the default playback device is the same as choosing, ‘System’ (this is left in for backward-compatibility).

To change the volume level of a specific recording device (microphone), select the, ‘Recording Device’ option and select the device you would like to modify. Choosing the, ‘Default’ device will always select the default playback device as indicated by Windows. If you want to change the volume of whatever device your speech engine is currently using, select, ‘Speech Engine Recording Device’ from the list.

Selecting the, ‘Active Window’ option will change the audio level of the application represented by Windows’ focused, active window.

To change the audio level of an application, select the application from the dropdown list. Note that this is a free input box and you can modify your selection (as detailed below). This box also respects tokens if needed.

The value in the drop-down box can contain wildcards indicated by asterisks (*). This

is handy when the title of the window changes. To indicate that the window title **contains** the value in the box, put an asterisk on each end. For instance, if you want to target any window that contains, 'VLC' in the title, put, '*****VLC*****' (without quotes) in the box. To indicate that the window title **starts with** the value in the box, put an asterisk at the end: 'VLC*'. To indicate that the window title **ends with** the value in the box, put an asterisk at the beginning: '*VLC'. The values are not case-sensitive. The first window found to match the criteria indicated will be selected. **Advanced**: Note that you can also use process names as they appear in the Windows Task Manager. You can use wildcards the same as you do with the window titles. Window titles are checked first, and then the process names. **More advanced:** If a window cannot be found by title or process name, the window class names will then be checked. Wildcards apply if you need them. Note that this will be the class name of the window itself and not the class name of a child control. Again, this is an advanced feature that you may never ever use.

Once you select the type of audio that you want to control, you can then set the audio's level using several methods. You can mute and unmute the audio by selecting either, 'Mute' or 'Unmute', or you and toggle the mute on and by selecting, 'Toggle Mute'. You can also set the level of the audio to a specific value by selecting, 'Level' and inputting a value. This value must resolve to a whole number from 0 to 100. So, if you want to set your volume to 50%, simply enter, '50' (without quotes) into the box. **Advanced:** Note that this box will also accept an integer variable name or a token that resolves into either a number or even an integer variable name.

To increase/decrease the current volume by a certain amount, select the, 'Offset' option and input the value by which you would like to increase or decrease the volume. For example, if you want to increase the volume by 10, input 10 in the input box. If you want to decrease the volume by 10, enter -10 in the input box. **Advanced:** Note that this box will also accept an integer variable name or a token that resolves into either a number or even an integer variable name.

Note: The application/active window volume is not the actual volume that is controlled by the selected application. For instance, there is a volume slider bar on Windows Media Player that controls the volume. VoiceAttack does NOT control that slider. VoiceAttack will control the volume for WMP as it is reflected in the System Volume Mixer. That means that if you set WMP's audio level value to 50, the slider in the System Volume Mixer will adjust the volume for WMP to 50% of whatever the system volume is currently set.

'Capture a Screenshot'

This action attempts to get grab a screenshot and store it in a place of your choosing. The, '**Source**' option lets you indicate if your screenshot is going to come from the active window, your primary screen, your entire desktop or from a region of your entire desktop. If you choose, 'Region', you'll need to supply the rectangular bounds that the capture is to select. X and Y are the top-left coordinates of this rectangle, with the origin (0, 0) being the top-left corner of your primary screen (that means that if you have a monitor to the left of your primary screen, the X-coordinate will be a negative

number). ‘Width’ and, ‘Height’ indicates how wide and high your rectangle will be, in pixels. Note that any one of the four, ‘Region’ boxes can take an integer variable or a token that can resolve to an integer value. If the variable or token cannot be resolved into an integer, the value will be considered zero.

The, ‘**Storage**’ option lets you indicate where the screenshot is to end up once it is captured. You can select the Windows clipboard (please be aware that that this will overwrite anything that’s already in your clipboard), a specific folder (the screenshot names will be automatically generated) or a specific file. Note that the folder and file options will accept any combination of tokens.

‘**Output File Type**’ lets you choose the image format of the captured screenshot. The resulting file size can vary depending on the image you are capturing, so, play around with this setting until you find the one that fits your needs. If your output file type is .jpg, you can select the quality level by changing the, ‘**Jpg Quality**’ option. This value can range from 0 to 100, with a value of 100 providing the best quality.

'Block/Unblock Keyboard Input'

This action works like a gaming aid and will allow you to have VoiceAttack attempt to block, unblock or toggle the blocking of keyboard key presses that you choose (you know… like when that annoying Windows key gets pressed all the time). ‘Attempt’ is indicated, as this will depend on your system. Factors such as other software running and the order by which you execute that software in regards to VoiceAttack can alter your experience. Also, if your game or app relies on input that cannot be blocked, VoiceAttack will not be able to block it. So, in a nutshell, ‘Your mileage may vary’.

Ok, back to it… There are three actions that you can perform:
Block - this blocks input from occurring from the selected keys.
Unblock - this will remove blocks that were previously put in place by a VoiceAttack, ‘Block’ action.
Toggle - this will either block unblocked input or unblock blocked input (it’s a toggle, lol).

The scope of blocking depends on the next option. If you want to just specify a certain number of keys, you can select the, ‘**Only the keys listed below**’ option. Then, simply press the keyboard keys that you want to block. The keys that are selected will appear in the box below the selection. Note that you can remove the keys by clicking on them or by clicking, ‘Clear’. When this action is executed, only the selected keys will be blocked. If you want to block ALL input *except* for certain keys, select the, ‘**All but the keys listed below**’ option. Note that indicating no keys with this option will block all keys.

Each, ‘Block Keyboard Input’ action is cumulative. What that means is that if you block, say, key ‘X’ in one action and block key, ‘Y’ in a second action, both, ‘X’ and ‘Y’ keys will be blocked at that point.

Note: Although key presses will continue to be executed by VoiceAttack actions,

VoiceAttack will not respond to key presses from the keyboard. Use with care.

'Block/Unblock Mouse Input'

This action works like a gaming aid and will allow you to have VoiceAttack attempt to block, unblock or toggle the blocking of various mouse actions that you choose. Again, just like in, 'Block/Unblock Keyboard Input', 'attempt' is indicated, as this will depend on your system (see previous section for rant ;)).

There are three actions that you can perform:
Block - this blocks input from occurring from the selected mouse actions.
Unblock - this will remove blocks that were previously put in place by a VoiceAttack, 'Block' action.
Toggle - this will either block unblocked input or unblock blocked input.

The scope of blocking depends on the next option. If you want to just specify certain actions, you can select the, '**Only the actions indicated below**' option. Then, just click on the items you want to block. You'll see all five standard mouse buttons (left, right, middle, back, forward) as well as four different scrolling actions (scroll forward, back, (tilt) left, (tilt) right). Note that you can remove the actions by clicking on them again or by clicking, 'Clear'. When this action is executed, only the selected actions will be blocked. If you want to block ALL mouse actions *except* for certain ones, select the, '**All but the actions indicated below**' option. Note that indicating no actions will block all selectable mouse actions.

Each, 'Block Mouse Input' action is cumulative. What that means is that if you block, say, the middle mouse button in one action and block the right mouse button in a second action, both the middle and right buttons will be blocked at that point.

Note: Although mouse actions will continue to be executed by VoiceAttack, VoiceAttack itself will not respond to your physical mouse actions. Use with care.

Note: Extended buttons, like the ones found on macro-enabled mice, can ONLY be blocked via the software that comes with those mice.

'Restrict Mouse Movement'

This action works like a gaming aid and will allow you to have VoiceAttack attempt to reduce the speed of the mouse. Again, just like in, 'Block/Unblock Keyboard/Mouse Input', 'attempt' is indicated, as this will depend on your system. Simply choose the level of movement restriction by using the slider. Choosing 0% clears any previously-invoked restriction, 10% would indicate only a small amount of movement restriction, while 100% would indicate a full block (the cursor will not move, so use with care).

Dictation (Speech to Text)

In order to be as flexible as possible, 'dictation' in VoiceAttack requires multiple parts.

Since VoiceAttack is already, 'listening' for your commands, you will need to be able to turn dictation on and off. There are two new actions to do this for you: Start Dictation Mode, and Stop Dictation Mode.

To turn on dictation mode, just add the action to one of your commands. You can initialize variables, play a sound, clear the dictation buffer (more on that later) or whatever you want before or after the Start Dictation action. For example, you can have a command called, 'Open Quote'. In that command, you can play a sound to notify you that dictation is on and, 'listening', followed by the, 'Start Dictation' action.

To turn off dictation mode, just add the Stop Dictation Mode action to a command you specify. Again, you can paste the dictation buffer, play a sound, clear variables or whatever in the same command.

Note: When dictation mode is on, you can still issue normal VoiceAttack commands. The commands will be processed and not included in the dictation buffer. That way, you can still, 'fire all weapons' when you're in the middle of messaging your wingmen ;)

The main part of the dictation feature is what is referred to as the speech buffer (or, 'the buffer', for short).

The speech buffer holds all the words that you are speaking. Every time you speak and then pause, the speech is converted to text and is added to the buffer. The buffer will continue to grow until you clear it. To clear the speech buffer, the, 'Clear Dictation Buffer' action is provided. Simply add the action to your specified command at the place where you want to clear the buffer. As an option in this action, you can limit what is cleared to the last statement in the buffer. Also, in the Start and Stop Dictation actions, you can optionally clear what is in the buffer.

The value in the dictation buffer is accessed in VoiceAttack in places that accept tokens (see, 'Tokens' in the VoiceAttack documentation for more info on how to use them... seriously... check it out because you can do a lot with tokens).
The tokens for the dictation buffer is {DICTATION} and {DICTATION:*options*}. For example, you can pop that token into the Text to Speech output box, into the Quick Input value box or into the Set Windows Clipboard value box.
VoiceAttack will convert the {DICTATION} token into what is contained in the buffer at the point that it is used.

Some things you might want to do with what is in the speech buffer:
-output to some application (messaging teammates)
-output to text to speech (feedback for you)
-output to a plugin, the VoiceAttack log, the Windows clipboard, etc.

There are two ways to get what is in the speech buffer out of VoiceAttack and into another application. One way is to output the text one character at a time using VoiceAttack's Quick

Input feature. This is a fairly reliable way to get your text out, however, it is one character at a time. That means that there will be a delay getting the text out. Since there is a delay, there are opportunities to accidentally hit other keys while the text is being output to your application.

The preferred way to get the buffered text out is by using the Windows clipboard. This way, all the text is output in one shot, reducing the chance for error. To add what is in the buffer to the Windows clipboard, simply add a, 'Set a Text Value to the Windows Clipboard' action to your command. In the, 'Value' box, just add the, '{DICTATION}' or '{DICTATION:*options*}' token (again, see, 'Tokens' in the VoiceAttack documentation for more info on how to use them). To paste from the Windows clipboard, simply issue your preferred method of pasting (CTRL + V or SHIFT + INSERT with appropriate delays).

Note: If you do not have access to the Windows clipboard (within games), outputting your text one character at a time may be your only option.

It also needs to be noted that dictation is only as reliable as your speech engine's training and/or your hardware and/or your configuration and drivers. It's not going to be nearly as accurate as your spoken commands. For some, dictation may be near flawless while others may find it hit or miss or even frustrating. Apologies in advance, however, this is a first go-round with dictation and is provided as-is (sorry!). Over time, I'm hoping that this will get even better.

'Start Dictation Mode'

This action will turn on VoiceAttack's, 'dictation' mode. Any recognized speech while this mode is active will be added to the dictation buffer. Recognized commands that are spoken on their own will still be processed first and not added to the dictation buffer (just in case you are dictating a message to your team and then you get attacked... lol). Note that dictation can be stopped and started and the dictation buffer will be maintained until you clear it.

The value in the dictation buffer can be accessed by using the {DICTATION} and {DICTATION:*options*} tokens. See more about this token in the Token section near the end of this document.

The, 'Clear dictation buffer before starting dictation mode' does just that. It clears the entire buffer once you start this action (for convenience... saves a step if you really need it).

'Stop Dictation Mode'

This action will turn off VoiceAttack's, 'dictation' mode. Note that the dictation buffer is still maintained while you start and stop dictation mode. That is, unless you select the, 'Clear dictation buffer after stopping dictation mode' option. Using this option will clear the dictation buffer immediately after stopping dictation mode. Again, more of a convenience feature to save steps.

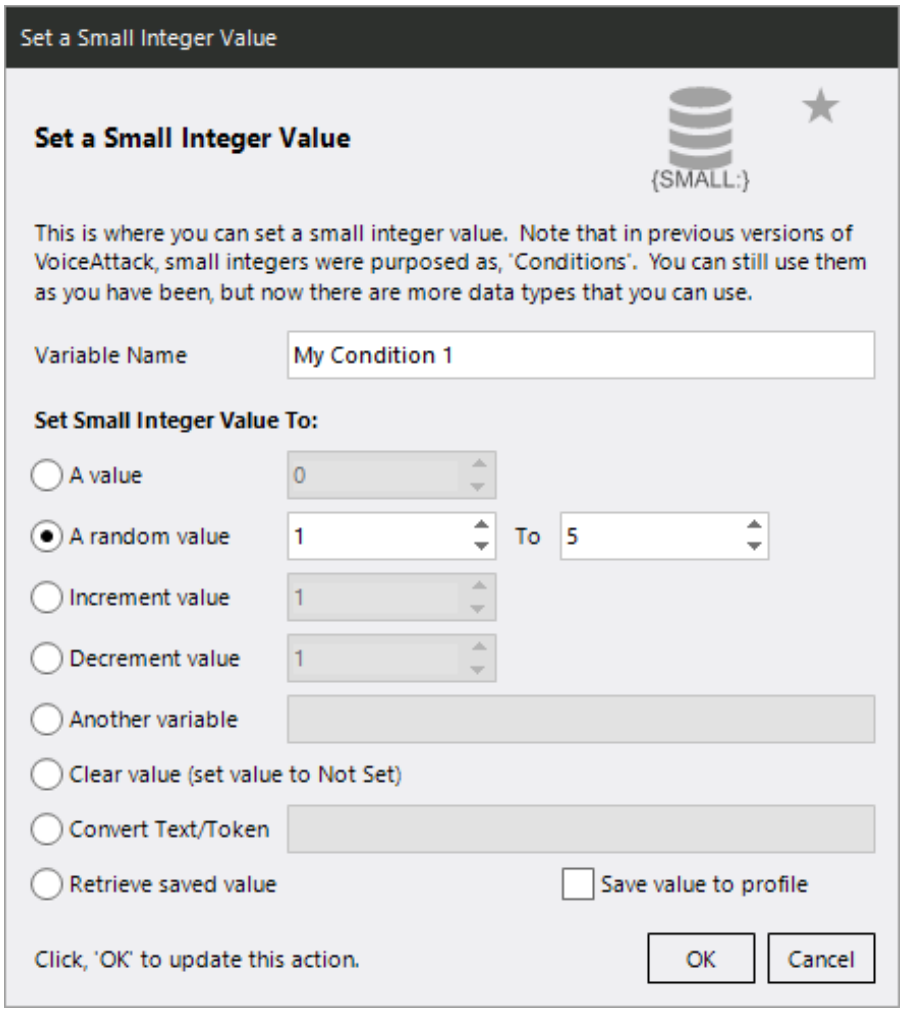
'Clear Dictation Buffer'

This action will clear the dictation buffer. The option, 'clear only the last statement' is handy if you botch the last thing you said (which will happen a lot, given the current state of the speech engine).

Advanced

The next set of command actions are considered, 'advanced' and are for use within VoiceAttack to allow users to make things a bit more flexible. There is a good chance that you will never use some of these features or use them very little (since they are a little bit outside of what you probably need to make VoiceAttack work for you). Have fun!

'Set a Small Integer (Condition) Value'



NOTE: Due to the expansion of VoiceAttack's available variable types, what used to be called a, 'Condition' has been renamed, 'Small Integer'. The Small Integer type is going to be left in for backward-compatibility, however, you may find the Integer type a little bit more useful. Make sure to check out the Integer, Boolean and Decimal variable types (and the accompanying new features) outlined below.

This command action will allow you to set the value of a small integer variable. These variables are used in conjunction with Conditional blocks ('If' statements) to control the flow of your command actions (more on conditional blocks below).

The Small Integer name can be whatever name you want. The variable names are not

case-sensitive and they must not contain semicolons or colons (variable names can only contain colons if contained within a token… this would be a good place to indicate that this input box also processes tokens).

The value that you set to your small integer variable can be explicit (you set an exact value), random (from a low value to a high value), incremental, decremental, to another small integer variable's value that you have defined, or to the value of text/tokens. The value can be no more than 32,767 and no less than -32,768. If you try to set your small integer variable to a value outside of this range, your small integer variable's value will be set to, 'Not set'.

If you want your small integer variable to be saved with the active profile, select the, 'Save value to profile' option. This will allow you to access the value between application sessions (VoiceAttack application is closed and then run again).

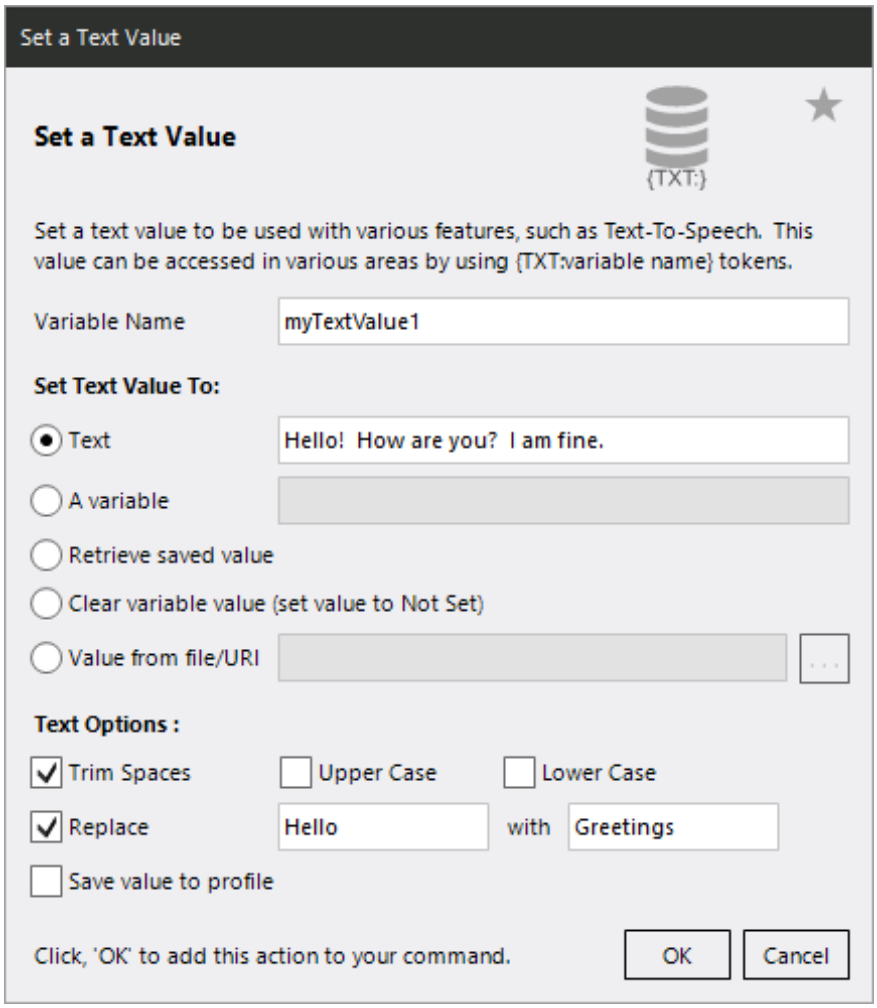
To retrieve the small integer variable saved with the profile, select the, 'Retrieve saved value' option.

Note: To clear all previously-saved small integer variables, see the, 'Clear Saved Values from Profile' action.

Note: You can define as many values as you want, however, the values that you define are not persisted. That is, they are not saved to disk. These values will be reset every time you restart VoiceAttack. If you want your values saved to disk for use between application sessions, select the 'Save value to profile' option.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for ease of use), however, for those that need a finer level of control, make sure to check out the, 'Advanced Variable Control (Scope)' section later in this document.

'Set a Text Value'



This command action will allow you to set a text value to a named variable. These values can be accessed by special tokens in various places within VoiceAttack (such as Text-To-Speech, Launch Application paths/parameters/working directories, etc).

The Text Value Name can be whatever name you want. This value will be stored at the application level (shared among all profiles), so care must be made in order to make your name is unique enough so you don't overwrite your values accidentally. The text value names are not case-sensitive and they must not contain semicolons or colons.

You can choose to set your text value explicitly, by typing in a value and selecting the, 'Text' option. This value can also contain other tokens that are replaced out when the command runs (see the section regarding tokens near the end of this manual).

You can also choose to set your text value to another text value variable. To do this, just select ‘A variable' option and type the name of the target text value variable in the box provided. Note that the variable can be the same as the one you are setting (for use if you just want to use the text options without assigning to another variable first).

To save the text variable value with the current profile, check the, 'Save value to profile'

box. The value will be available even if the VoiceAttack application is closed and opened again. This is a simple way to save your text value variable to disk.

To retrieve the saved value, select the, 'Retrieve saved value' option.

To set the text value variable to an unset state, select the, 'Clear value' option.

If you want to get the value for your text value variable from a file or URL, select the, 'Value from file/URI' option. To browse for a file, click the button with the ellipsis ('…'). To get the value from a URL, just type the address. For example, you can try, 'http://www.voiceattack.com/test.htm' (without the quotes). VoiceAttack will attempt to get the text from the response. Note: VoiceAttack reads data from these sources using UTF-8 encoding.

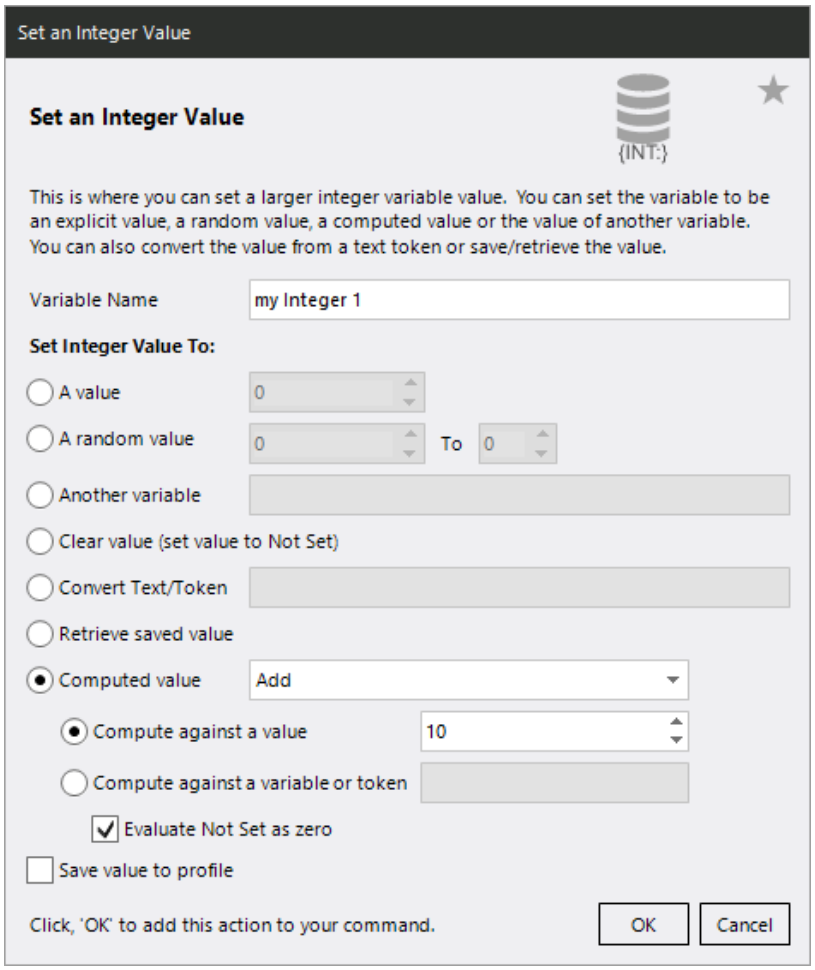
To access the values stored as text values, you will use the {TXT:valueName} token (see the section about tokens near the end of this manual).

There are a few additional options available to you when you set a text variable. The options are Trim Spaces, Upper Case, Lower Case and Replace. These are applied after the variable is set. Trim Spaces will remove any spaces/whitespace from either end of the text value. If your text value is, ' Hello, how are you? ', using the Trim Spaces option will update the value to be, 'Hello, how are you?'. The Upper Case and Lower Case options will convert the text values to either all upper or all lower case characters. Using the Replace option will allow you to replace a portion of the text value with another value. For instance, you can replace, 'Hello' with 'Greetings' by putting 'Hello' in the first box and 'Greetings' in the second box. Any instance of the word, 'Hello' in the text value will be replaced with, 'Greetings'. Note that, 'Replace' is case-sensitive, and both of the 'Replace' input boxes can accept tokens.

Note: You can define as many values as you want, however, the values that you define are not persisted. That is, they are not saved to disk (unless you check the, 'save values to profile' option). These values will be reset every time you restart VoiceAttack.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for good reason), however, for those that need a finer level of control, make sure to check out the, ‘Advanced Variable Control (Scope)’ section later in this document.

'Set an Integer Value'



This command action will allow you to set the value of an integer variable. These variables can be used in conjunction with Conditional blocks ('If' statements) to control the flow of your command actions, provide feedback through things like text-to-speech as well as do other stuff like provide information to plugins.

The Integer Variable Name can be whatever name you want. This value will be stored at the application level (shared among all profiles), so care must be made in order to make your name is unique enough so you don't overwrite your values accidentally. The variable names are not case-sensitive and they must not contain semicolons or colons (variable names can only contain colons if contained within a token… this would be a good place to indicate that this input box also processes tokens).

The purpose of this screen is to set the value of the variable, and you do that by selecting one of several different ways. The first way is to set an exact value (such as 500). Just select the option labeled, 'A value' and type the value in the box.
Another way to set an integer value is to give it a random value. Just select, 'A random value' and provide a minimum and maximum value and the variable will have a random number chosen within that range.

You can set your variable to the same value as another variable. Select, 'Another

variable' and type the name of the variable with the value that you want copied in the box provided.

To clear the value of your variable (make the value be, 'Not set' (programmers will call this, 'null'), select the, 'Clear value' option.

If you have a value in text or in a token, you can attempt to convert the value to an integer by selecting, 'Convert Text/Token' and type the text and/or tokens into the box provided. If the value cannot be converted, the variable value will be, 'Not set'. This is kind of advanced, and you may never even use this option.

If you want your integer variable to be saved with the active profile, select the, 'Save value to profile' option (check box at the bottom). This will allow you to access the value between application sessions (VoiceAttack application is closed and then launched again).

To retrieve the integer variable saved with the profile, select the, 'Retrieve saved value' option. If the value was previously saved (as indicated above), the value will be set. If no value is available, the variable value will be, 'Not set'.
Note: To clear all previously-saved integer variables, see the, 'Clear Saved Values from Profile' action.

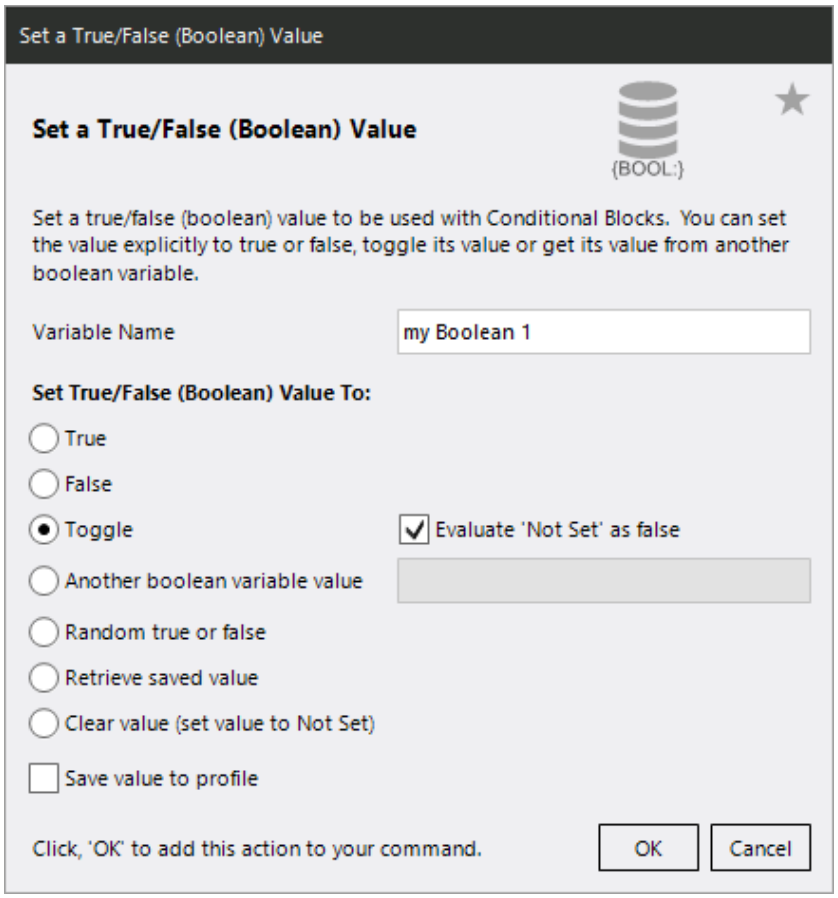
If you want your integer variable's value to be computed for you, there are some simple math functions available to you. First, select the, 'Computed value' option. Next, select the appropriate function. You can add, subtract, multiply and divide (with integer division… 7 divided by 3 is 2, for example), or get the remainder (Modulus)… 7 Mod 3 is 1, for example). The next thing you will want to do is indicate what you would like to compute against… that can be an explicit value (such as 2) or another variable or even a converted token. To select an explicit value, select, 'Compute against a value' and provide a value. To select another variable, choose, 'compute against a variable or token' and provide the variable name. To compute against a token, select this same option and indicate the token in the box provided. If the token cannot be converted, or the computed value falls outside of valid range (that is, between -2147483648 and 2147483647) the value of computation will be, 'Not set'.

As a convenience feature, there is a check box labeled, 'Evaluate Not set as zero'. This will allow you to initialize your variables as zero if they are not set when computing values. This is merely to save an initialization step (yes, another advanced bit you may never use).

Note: You can define as many values as you want, however, the values that you define are not persisted. That is, they are not saved to disk. These values will be reset every time you restart VoiceAttack. **If you want your values saved to disk** for use between application sessions, select the 'Save value to profile' option.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for good reason), however, for those that need a finer level of control, make sure to check out the, ‘Advanced Variable Control (Scope)’ section later in this document.

'Set a True/False (Boolean) Value'



This command action will allow you to set the value of True/False (Boolean) variable. These variables can be used in conjunction with Conditional blocks ('If' statements) to control the flow of your command actions, provide feedback through things like text-to-speech as well as do other stuff like provide information to plugins.

The Variable Name can be whatever name you want. This value will be stored at the application level (shared among all profiles), so care must be made in order to make your name is unique enough so you don't overwrite your values accidentally. The variable names are not case-sensitive and they must not contain semicolons or colons (variable names can only contain colons if contained within a token… this would be a good place to indicate that this input box also processes tokens).

The purpose of this screen is to set the value of the variable, and you do that by selecting one of several different ways. The first way is to set an explicit value (such as either True or False). Just select the option labeled, 'True' or the option labeled, 'False' to set your variable accordingly.

Another way to set a Boolean variable is to toggle its value. So, if a variable's value is True, it will be set to False. If it is False, it will be set to True.
Note: A Boolean variable that is, 'Not set' (programmers call this, 'null') will not toggle. The value will remain, 'Not set'. You can set the, '**Evaluate 'Not set' as false**' option to treat null (Not set) variable values as false. This may save a few steps here and

there.

You can set your variable to the same value as another variable. Select, 'Another Boolean variable value' and type the name of the variable with the value that you want copied in the box provided.

If you would like to set your Boolean variable to a random true or false value (like a coin toss), select, ‘Random true or false’.

To clear the value of your variable (make the value be, 'Not set' (programmers will call this, 'null'), select the, 'Clear value' option.

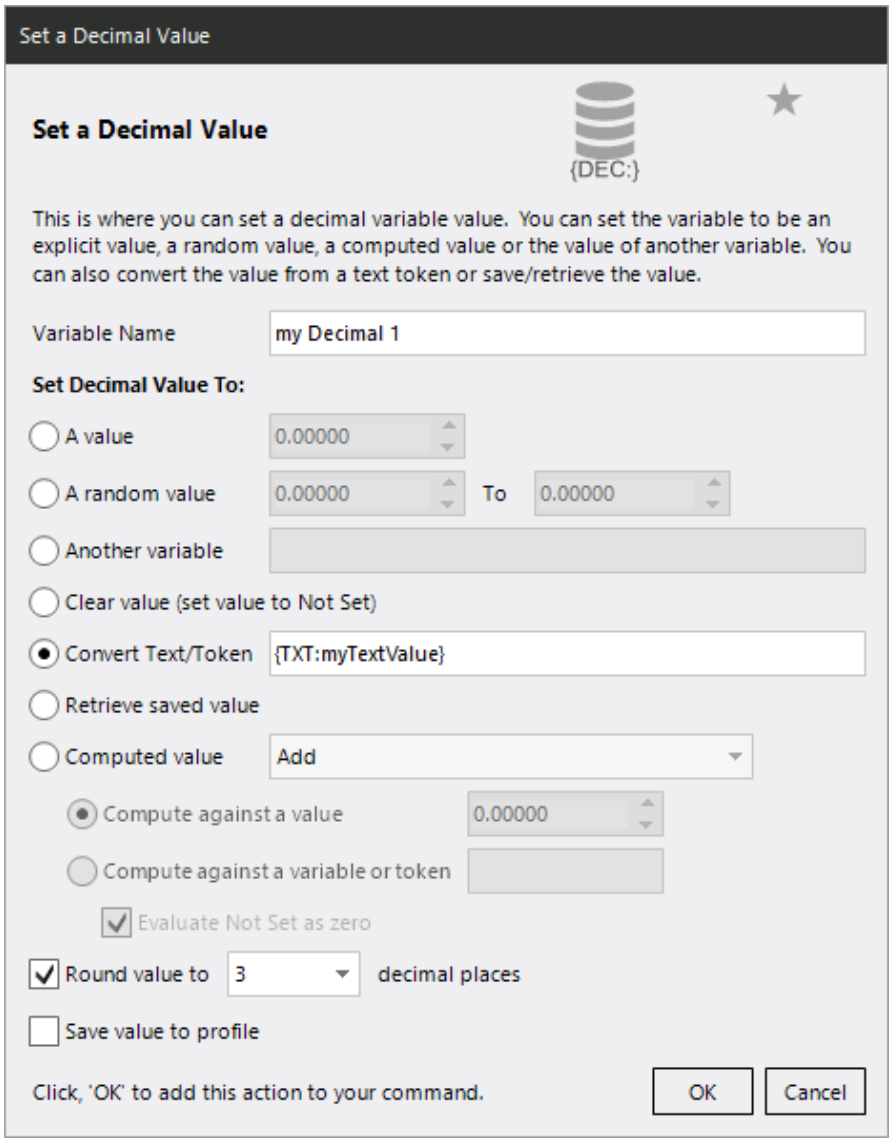
If you want your Boolean variable to be saved with the active profile, select the, 'Save value to profile' option (check box at the bottom). This will allow you to access the value between application sessions (VoiceAttack application is closed and then launched again).

To retrieve the Boolean variable saved with the profile, select the, 'Retrieve saved value' option. If the value was previously saved (as indicated above), the value will be set. If no value is available, the variable value will be, 'Not set'.
Note: To clear all previously-saved Boolean variables, see the, 'Clear Saved Values from Profile' action.

Note: You can define as many values as you want, however, the values that you define are not persisted. That is, they are not saved to disk. These values will be reset every time you restart VoiceAttack. **If you want your values saved to disk** for use between application sessions, select the 'Save value to profile' option.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for good reason), however, for those that need a finer level of control, make sure to check out the, ‘Advanced Variable Control (Scope)’ section later in this document.

'Set a Decimal Value'



This command action will allow you to set the value of a decimal variable. These variables can be used in conjunction with Conditional blocks ('If' statements) to control the flow of your command actions, provide feedback through things like text-to-speech as well as do other stuff like provide information to plugins.

The Variable Name can be whatever name you want. The variable names are not case-sensitive and they must not contain semicolons or colons (variable names can only contain colons if contained within a token… this would be a good place to indicate that this input box also processes tokens).

The purpose of this screen is to set the value of the variable, and you do that by selecting one of several different ways. The first way is to set an exact value (such as 100.1). Just select the option labeled, 'A value' and type the value in the box.

Another way to set a decimal variable value is to give it a random value. Just select, 'A

random value' and provide a minimum and maximum value and the variable will have a random number chosen within that range.

You can set your variable to the same value as another variable. Select, 'Another variable' and type the name of the variable with the value that you want copied in the box provided.

To clear the value of your variable (make the value be, 'Not set' (programmers will call this, 'null'), select the, 'Clear value' option.

If you have a value in text or in a token, you can attempt to convert the value to a decimal by selecting, 'Convert Text/Token' and type the text and/or tokens into the box provided. If the value cannot be converted, the variable value will be, 'Not set'. This is kind of advanced, and you may never even use this option.

If you want your decimal variable to be saved with the active profile, select the, 'Save value to profile' option (check box at the bottom). This will allow you to access the value between application sessions (VoiceAttack application is closed and then launched again).

To retrieve the decimal variable saved with the profile, select the, 'Retrieve saved value' option. If the value was previously saved (as indicated above), the value will be set. If no value is available, the variable value will be, 'Not set'.
Note: To clear all previously-saved decimal variables, see the, 'Clear Saved Values from Profile' action.

If you want your decimal variable's value to be computed for you, there are some simple math functions available to you. First, select the, 'Computed value' option. Next, select the appropriate function. You can add, subtract, multiply and divide (plus several more functions). The next thing you will want to do is indicate what you would like to compute against… that can be an explicit value (such as 2.6) or another variable or even a converted token. To select an explicit value, select, 'Compute against a value' and provide a value. To select another variable, choose, 'compute against a variable or token' and provide the variable name. To compute against a token, select this same option and indicate the token in the box provided. If the token cannot be converted, or the computed value falls outside of the acceptable range of values for a decimal (-79228162514264337593543950335 to 79228162514264337593543950335 (lol)) , the value of computation will be, 'Not set'.

As a convenience feature, there is a check box labeled, 'Evaluate Not set as zero'. This will allow you to initialize your variables as zero if they are not set when computing values. This is merely to save an initialization step (yes, another advanced bit you may never use).

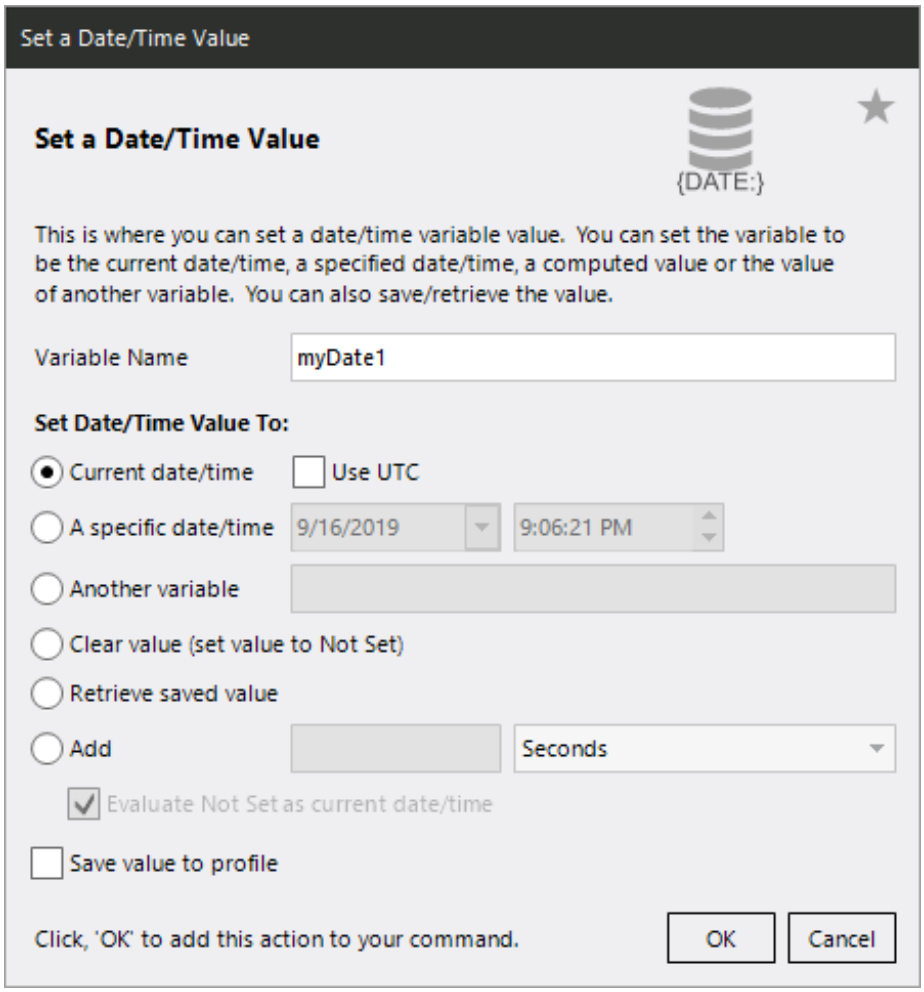
If you would like your assigned variable to be rounded to a certain number of decimal places (from 0 up to 10), select the 'Round value' option and choose the appropriate value.

Note: You can define as many values as you want, however, the values that you

define are not persisted. That is, they are not saved to disk. These values will be reset every time you restart VoiceAttack. **If you want your values saved to disk** for use between application sessions, select the 'Save value to profile' option.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for ease of use), however, for those that need a finer level of control, make sure to check out the, ‘Advanced Variable Control (Scope)’ section later in this document.

'Set a Date/Time Value'



This command action will allow you to set the value of a date/time variable. These variables can be used in conjunction with Conditional blocks ('If' statements) to control the flow of your command actions, provide feedback through things like text-to-speech as well as do other stuff like provide information to plugins.

The Variable Name can be whatever name you want. This value will be stored at the application level (shared among all profiles), so care must be made in order to make your name is unique enough so you don't overwrite your values accidentally. The variable names are not case-sensitive and they must not contain semicolons or colons (variable names can only contain colons if contained within a token… this would be a good place to indicate that this input box also processes tokens).

The purpose of this screen is to set the value of the variable, and you do that by selecting one of several different ways. The first way is to set the date/time variable to the current date/time (the current date/time when the action is executed). Just select the option, 'Current date/time'. If you want to set this value to UTC, select the, 'Use UTC' option.

Another way is to set an exact date and time. You can do this by selecting the,

'Specific date/time' option and selecting the date and time in the boxes provided.

You can set your variable to the same value as another variable. Select, 'Another variable' and type the name of the variable with the value that you want copied in the box provided.

To clear the value of your variable (make the value be, 'Not set' (programmers will call this, 'null'), select the, 'Clear value' option.

If you want your date/time variable to be saved with the active profile, select the, 'Save value to profile' option (check box at the bottom). This will allow you to access the value between application sessions (VoiceAttack application is closed and then launched again).

To retrieve the date/time variable saved with the profile, select the, 'Retrieve saved value' option. If the value was previously saved (as indicated above), the value will be set. If no value is available, the variable value will be, 'Not set'.
Note: To clear all previously-saved date/time variables, see the, 'Clear Saved Values from Profile' action.

If you want add or subtract time from your date/time variable, there are some simple options available. Select the, 'Add' option and then indicate the number of seconds, milliseconds, minutes, hours, days, months or years to add to the variable. To subtract time from your date/time variable, simply use negative values in the input box. Note that this input box can contain a large integer variable, as well as any combination of tokens that may resolve into an integer. If your date/time variable is, 'Not set', adding or subtracting time from it will still result in, 'Not set'. As a convenience, the, 'Evaluate Not set as current date/time' option is available. If this option is selected and the variable is new or cleared (Not set), the variable will initialize as the current date/time (might save a step).

Note: You can define as many values as you want, however, the values that you define are not persisted. That is, they are not saved to disk. These values will be reset every time you restart VoiceAttack. **If you want your values saved to disk** for use between application sessions, select the 'Save value to profile' option.

Advanced: Variables can be scoped at the command-level, at the profile-level and globally. Most will use the globally-scoped variables (for good reason), however, for those that need a finer level of control, make sure to check out the, ‘Advanced Variable Control (Scope)’ section later in this document.

'Convert a Value'

This action will allow you to convert the value from one variable type to another. For instance, if you have a text variable with a value of “1234” and you want that text to be converted to an integer value, you can select what integer variable in which to place the value of 1234 into. The available types to convert are the six standard types: text, small integers, decimals, integers, true/false (Boolean) and date/time.

Simply put the source and target variable names into the source variable and target variable boxes and select their types. Note that the source and target variable boxes can process tokens to resolve variable names.

Note - Any failed attempt to convert a value will result in the target variable having its value removed (not set).

- Converting text to any other type will attempt to parse the text into the target type, EXCEPT Boolean, which will also accept “0” as false and “1” as true in addition to “true” and “false”.

- Converting a small integer, decimal or small integer into a Boolean will normally fail EXCEPT if the value is 0 or 1. If the value is 0, the converted Boolean value will be false. If the value is 1, the Boolean value will be true.

- The only type that can be converted to a date/time is text (or another date/time).

- Converting from a Boolean to text will yield the text value for the Boolean (and not “0” or “1”).

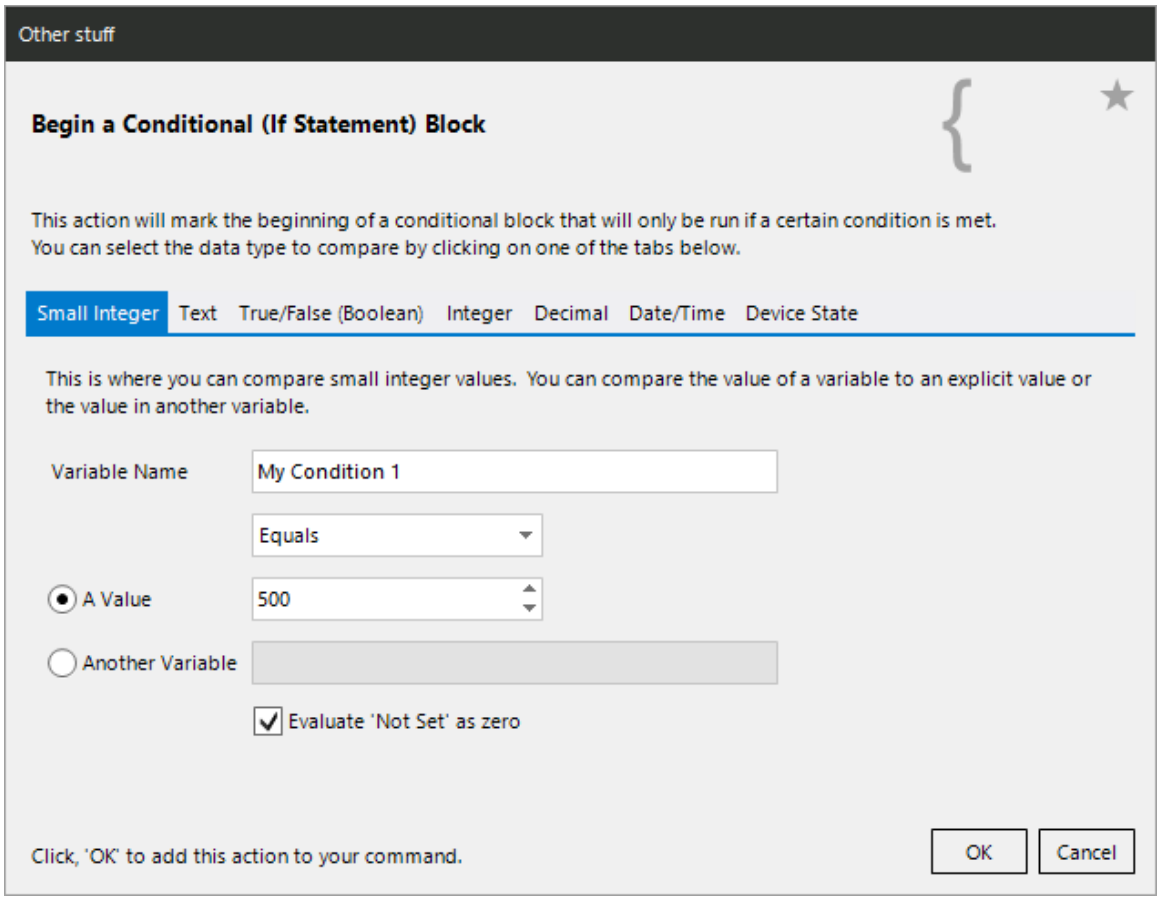
'Begin a Conditional (If Statement) Block'

This command action is what you will use to begin a conditional block. Think of a conditional block as a simple, 'IF' statement. This block MUST be used with a corresponding End Conditional Block action (below).

Basically, the command actions that occur between the Begin and End Conditional Blocks will ONLY be executed if the comparison you define meets the criteria that you indicate in the Begin Conditional Block.

You will need to first indicate what type of comparison you are going to make. If you are going to compare small integers, select the, 'Small Integer Tab'. If you are going to compare the text in a text variable, select the, 'Text' tab (and so on for the other data types).

Small Integer (Condition) Value Comparison



If you are wanting to compare small integer (formerly called, 'conditions') values and chose this tab, the next step is to indicate what variable you are going to check by typing its name in the Variable Name field (you would have set this value in the, 'Set a Small Integer (Condition) Value' action... see above).

Next, you will need to establish how the variable is going to be compared, by

dropping down the, 'operator' box. You can choose Equals, Does Not Equal, Greater Than, Less Than, Is Set and Is Not set (a variable is, 'Set' if a value has actually been assigned to the variable. In programming-speak, the value would be considered null or not null).

The last thing you will need to do is to indicate the value that you are going to compare the variable to. This can be an explicit value (by selecting, 'A Value' and filling in the box), or the value of another small integer variable that you had set up (by selecting, 'Another Variable' and typing the name of that variable in the box provided).

Optionally, you can select the, “Evaluate, ‘Not set’ as zero” feature that will automatically evaluate the included variables (variables indicated in both, ‘Variable Name’ as well as, ‘Another Variable’) as zero when comparing. If you want to compare variables as null (‘Not set’) when they are null (‘Not set’), make sure to uncheck this option.

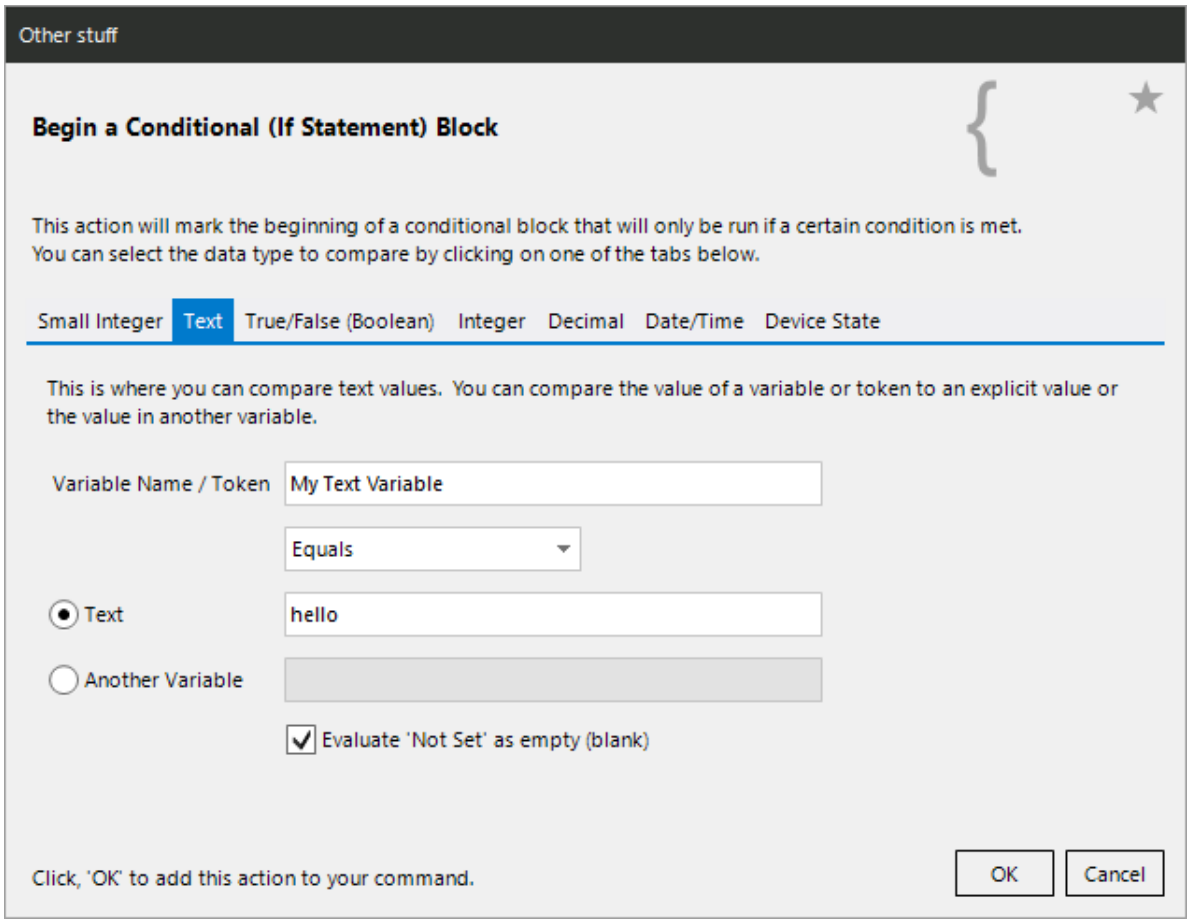
If the comparison is made and the condition is met, the command action immediately following the Begin a Conditional Block action will be executed. If the condition is NOT met, the command action immediately following a corresponding Else or the corresponding End Conditional Block will be executed (this is why the End Conditional Block is required).

Note: If a variable that is being compared is not set, the comparison will always result as false. So, if 'My Condition 1' is not set, and you try to compare it to 0 by setting the operator as, 'not equal to', the result will be false and the code will continue after the end block (see information above regarding evaluating, ‘Not set’ as empty (blank)).

Note: To create a simple, single condition, ‘If’ statement, choose the, ‘Single Condition’ option from the menu. To create a compound, ‘If’ statement (that is, an ‘If’ statement that contains multiple conditions (‘and’ and ‘or’), select the, ‘Compound Condition Builder’ option from the menu. For more information regarding compound conditions, see the section titled, ‘Using the Condition Builder’ later in this document.

Also note that you can convert a single conditional statement to a compound statement by right clicking on the action on the command screen and choosing the, ‘Edit with condition builder’ option.

Text Value Comparison



If you are wanting to compare a text variable and chose this tab, the next step is to indicate what text variable you are going to check by typing its name in the, 'Variable Name / Token' field (you would have set this value in the, 'Set a Text Value' action… see way below). Note that this box can accept tokens as well. This way, you can compare either variable values or the value of a rendered text token.

Next, you will need to establish how the variable is going to be compared, by dropping down the, 'operator' box. You can choose Equals, Does Not Equal, Starts With, Does Not Start With, Ends With, Does Not End With, Contains, Does Not Contain, Is Set and Is Not set (a text variable is, 'Set' if a value has actually been assigned to the variable. If you are a programmer, you would refer to this as the variable being null or not null).

The last thing you will need to do is to indicate the value that you are going to compare the text variable to. This can be an explicit value (by selecting, 'Text' and filling in the box), or the value of another text variable that you had set up (by selecting, 'Another Variable' and typing the name of that variable in the box provided).

Optionally, you can select the, “Evaluate, ‘Not set’ as empty (blank)” feature that will automatically evaluate the included variables/tokens (variables indicated in both, ‘Variable Name/Token’ and ‘Another Variable’, as well as the converted value of what is

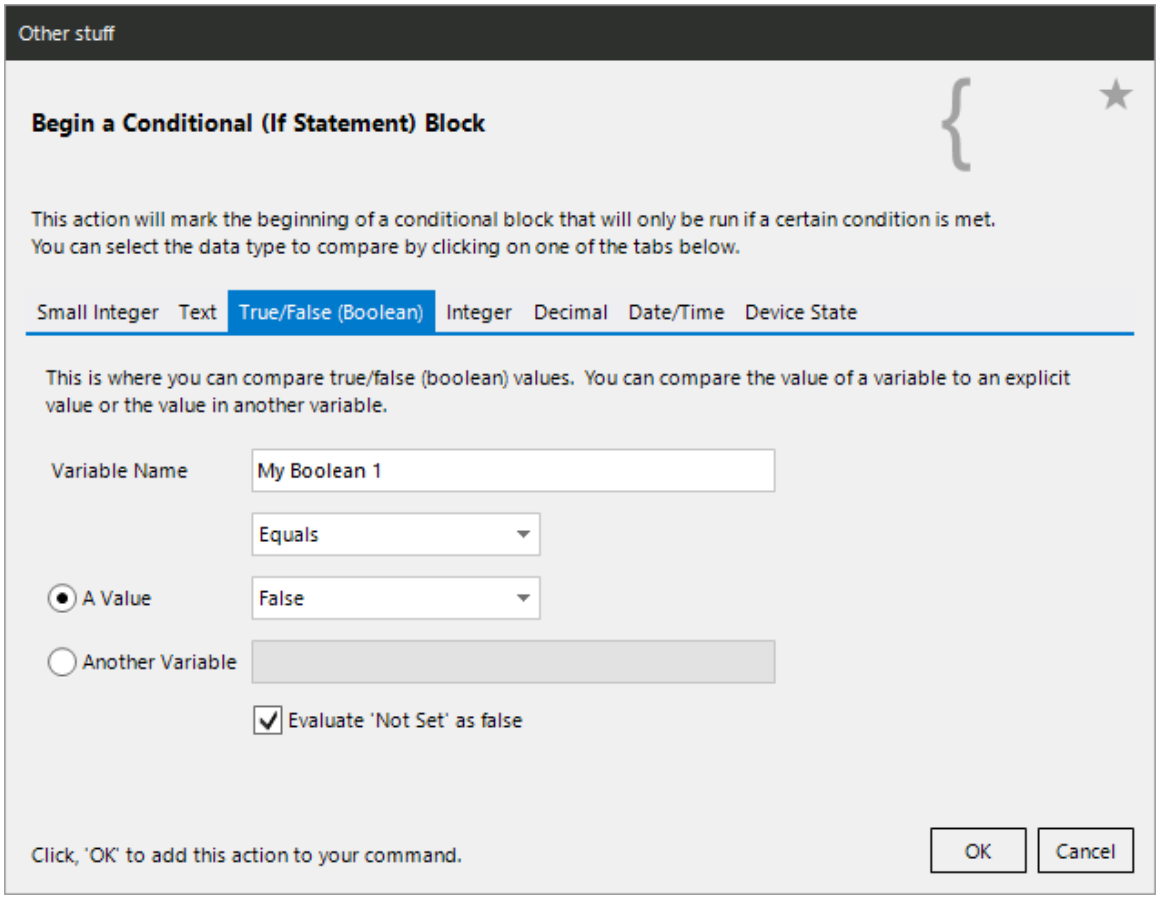
placed in the Token field) as empty (blank) when comparing. If you want to compare variables as null (‘Not set’) when they are null (‘Not set’), make sure to uncheck this option.

If the comparison succeeds, the command action immediately following the Begin a Conditional Block action will be executed. If the comparison fails, the command action immediately following a corresponding Else or the corresponding End Conditional Block will be executed (this is why the End Conditional Block is required).

Note: If a text value variable that is being compared is not set, the comparison will always result as false (see information above regarding evaluating, ‘Not set’ as empty (blank)).

Note: The 'Text' option also accepts tokens.

True/False (Boolean) Comparison



If you are wanting to compare a true/false (Boolean) variable and chose this tab, the next step is to indicate what Boolean variable you are going to check by typing its name in the, 'Variable Name' field (you would have set this value in the, 'Set a True/False (Boolean) Value' action… see way below).

Next, you will need to establish how the variable is going to be compared, by dropping down the, 'operator' box. You can choose Equals, Does Not Equal, Is Set and Is Not set (a Boolean variable is, 'Set' if a value has actually been assigned to the variable. If you are a programmer, you would refer to this as the variable being null or not null).

The last thing you will need to do is to indicate the value that you are going to compare the Boolean variable to. This can be an explicit value (by selecting, True or False from the 'Value' field), or the value of another Boolean variable that you had set up (by selecting, 'Another Variable' and typing the name of that variable in the box provided).

Optionally, you can select the, “Evaluate, ‘Not set’ as false” feature that will automatically evaluate the included variables (variables indicated in both, ‘Variable Name’ as well as, ‘Another Variable’) as false when comparing. If you want to compare variables as null (‘Not set’) when they are null (‘Not set’), make sure to uncheck this option.

If the comparison succeeds, the command action immediately following the Begin a

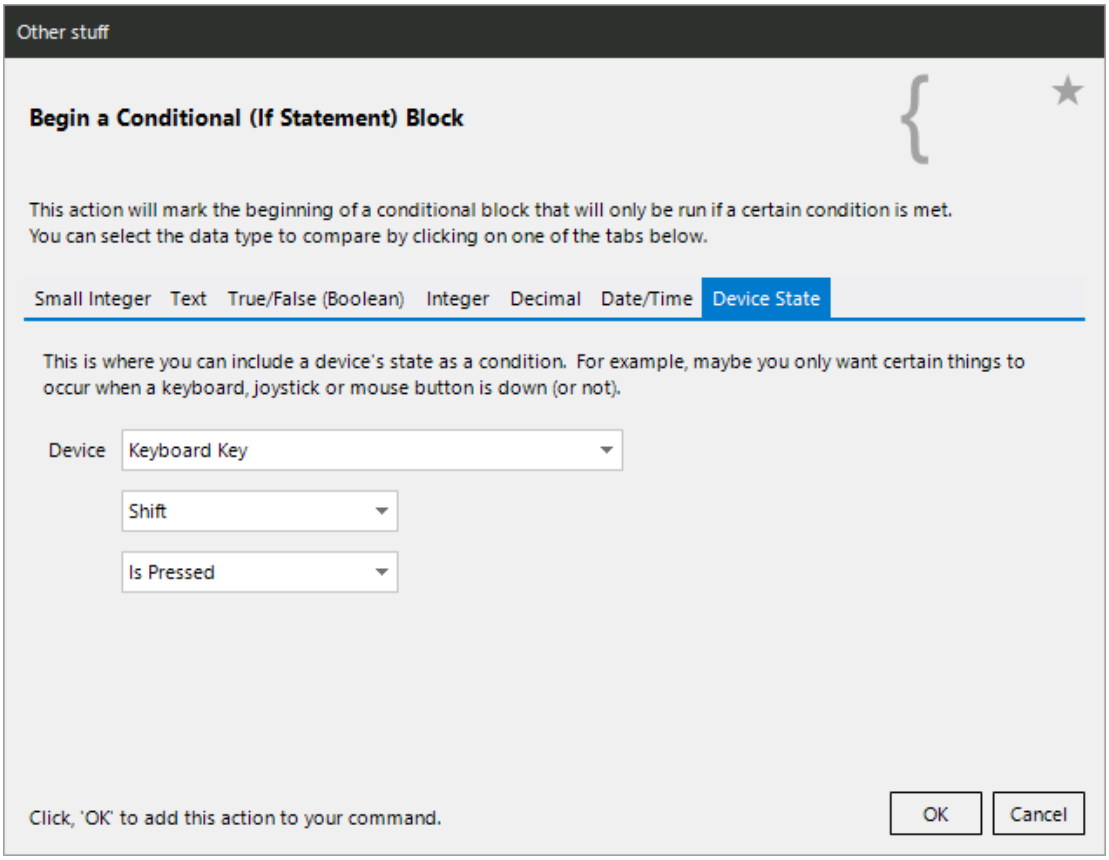
Conditional Block action will be executed. If the comparison fails, the command action immediately following a corresponding Else or the corresponding End Conditional Block will be executed (this is why the End Conditional Block is required).

Note: If a True/False (Boolean) variable that is being compared is not set, the comparison will always result as false (see information above regarding evaluating, 'Not set' as empty (blank)).

Integer, Decimal and Date/Time Value Comparison

If you would like to compare the values of integers, decimal and date/time variables, simply select the appropriate tab. The functionality of these three options are basically the same as the Small Integer comparisons indicated above (so, we'll save a tree and not duplicate even more stuff... lol.).

Device State



The Device State condition allows you to check the state of your keyboard keys, mouse buttons and joystick buttons and have your command be able to react to these states. For instance, you may want to check if a certain key is down when you say, 'Fire Weapons'. Your command could do something differently if the, 'X' key is down rather than if it is not. Also, you can check the state of your mouse and/or joystick

buttons. So, your, ‘Fire Weapons’ command could behave totally differently if Shift, Right Mouse Button and Joystick Button 7 are down ;) Simply choose the device you would like to check, then select the key, button or position. Next, select whether you want to check if the button/key is pressed or not pressed. Note that you can also check to see if the key/button you are checking is the ONLY key/button pressed, by selecting the, ‘Only key/button pressed’ option. To do a device-wide check to see if any key/button is pressed at all, select the, ‘<Any key/button>’. To do a device-wide check to see if no keys/buttons are pressed, select the, ‘<No key/button>’ option.

Mouse and joystick positions can also be checked. So, if your mouse enters an area of the screen, or if your joystick is pushed to the limit, VoiceAttack can be set up to react. Simply choose the device and position aspect you would like to check and indicate the value in the box at the bottom. Note that this box can contain integer values, variable names of variables that resolve to an integer, tokens that resolve to an integer or tokens that resolve to variable names that resolve to an integer (whew). Note that the mouse has two different aspects: screen and app/window. Use, ‘screen’ when you want your coordinates to relate to the entire screen ((0,0) would be the top-left corner of the screen). Use, ‘app/window’ when you want the coordinates to relate to the active window ((0,0) would be the top-left corner of the active window). Also note that the value of the joystick positions will be -1 if the joystick position cannot be accessed.

'Compound Condition Builder'

This action works similarly to the previously-mentioned, 'Begin a Conditional (If Statement) Block' screens, with the difference being that multiple (compound) conditions can be used to build a Conditional (If) Statement Block or Loop Start. There is a lot to this screen, so in order to keep this description short, please see the section, ‘**Using the Condition Builder**’ section later in this document.

'Add Else If to a Conditional Block'

This action will allow you to add an, 'Else If' to your condition block. That is, if the result of the beginning, 'If' statement is false (condition not met), you can use an Else If block to do another comparison. You can have as many, 'Else If' blocks as you wish between the Begin and End Conditional block actions. The options for the, 'Else If' are the same as what you find in the 'Begin a conditional (if statement) block'. If all the 'Else If' blocks do not have their conditions met, the command will then go to an existing, 'Else' action or to the End (if there are no, 'Else' actions for the containing conditional block).

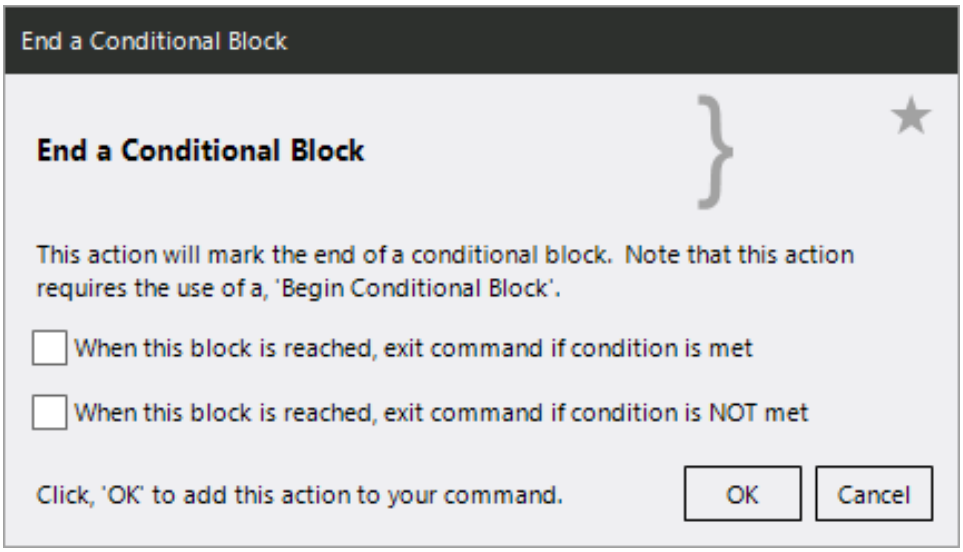
To create a simple, single condition, ‘Else If’ statement, choose the, ‘Single Condition’ option from the menu. To create a compound, ‘Else If’ statement (that is, an, ‘Else If’ statement that contains multiple conditions), select the, ‘Compound Condition Builder’ option from the menu. For more information regarding compound conditions, see the section titled, ‘Using the Condition Builder’ later in this document.

'Add Else to a Conditional Block'

You can add an 'Else' action to direct the flow of your condition block. If the result of the conditional block is true, all actions in the conditional block ABOVE the, 'Else' will be executed. When the actions between the start of the conditional block and the, 'Else' are finished, the command will jump down to (and execute) the, 'End Conditional Block' action.

When a conditional block does not meet the requirements (result is false), all actions between the, 'Else' action, up to and including the 'End Conditional Block' action will be executed.

'End a Conditional Block'



This command action is what you will use to end a condition block. This MUST be used in conjunction with a Begin Condition Block action (see above).

When a Condition Start Block action is executed, a value is checked to see if a condition is met. If the condition is met, all the actions between the Condition Start Block and the End Condition block are executed. If the condition is not met, the code following the End Condition Block is executed (everything between the Condition Start and Condition End blocks is skipped).

If you select either of the options indicated in the End Condition block, you can have one more check before processing resumes after the block. You have the option of completely exiting the command if the condition is or is not met. One thing this is useful for is to bypass a long list of condition checks.

A sample conditional block with Else If and Else actions is below:

Set 'myTextVariable' to 'howdy'
Begin Text Compare 'myTextVariable' equals, 'hello'

Say, 'You said hello'

Else If 'myTextVariable' equals, 'greetings'

Say, 'You said greetings'

Else If 'myTextVariable' contains, 'howdy'

Begin Text Compare 'myTextVariable' contains 'partner'

Say, 'You said howdy and I'm not your partner'

Else

Say, 'You said howdy'

End Condition

Else

Say, 'You didn't say anything I recognize'

End Condition

In this example, TTS would say, 'You said howdy'. If, 'myTextVariable' was set to 'hola', TTS would say, 'You didn't say anything I recognize'. Note that the conditional blocks can contain other conditional blocks (nested).

'Add a Loop Start'

There are a few options when adding loops: Single-condition, 'while' loops, compound-condition, 'while' loops, and loops that repeat a certain number of times.

- Loop Start - Single Condition (While Loop)

  This action indicates the beginning of a looping block that will continue looping as long as the indicated condition is met. The options for the loop condition are identical to the ones found in '**Begin a Conditional (If Statement) Block**' (see above), so, they will not be repeated here (there's a lot of options (lol)). All actions between the Loop Start and Loop End will be repeated. Once the condition is not met, the command flow will go to the command action immediately following the **Loop End** (see below). Note that loops can contain other loops (nested loops).

- Loop Start - Compound Condition (While Loop)

  This action works similarly to the previously-mentioned, '**Begin a Conditional (If Statement) Block**' screens, with the difference being that multiple (compound) conditions can be used to build a Loop Start. There is a lot to this screen, so in order to keep this description short, please see the section, '**Using the Condition Builder**' section later in this document.

- Loop Start - Repeat a Certain Number of Times (For Loop)

  This action has two different options. The first and easiest option is to repeat a block a specified number of times. The second, more advanced option is to repeat a block a number of times using a range (and optional indexer).

  Using the first option labeled, '**Number of times**', you can simply specify how many times you would like a block of actions repeated. In the input box, you

can specify the whole number to indicate the number of times to repeat the loop block. This box also accepts integer (large) variables. When the loop start is first encountered, the value in the variable is resolved and if it is valid, it is used to indicate how many times the block will be repeated. This box also accepts any combination of tokens. If the tokens render into a valid whole number, that value will be used. If the tokens render into a valid variable name, the variable will be resolved and its value used. **Note**: The value can be a negative number and still be valid. So, if the value is -3, the loop will repeat 3 times. Hint: If you need to bail out of this type of loop at any time, try using, '**Jumps**' and '**Jump Markers**'.

The second option is labeled as, '**Range (For Loop)**'. This option most resembles what is known as a, 'for' loop in programmer-speak. The loop will repeat from the value indicated in the, 'From' box (inclusive) to the value indicated the 'To' box (inclusive) (Note that these two boxes are also overloaded to work just like the 'Number of times' box (above)). So, if you put a value that resolves to 3 in the, 'From' box and a value that resolves to 7 in the, 'To' box, the block will repeat 5 times (remember, the values are inclusive). Note that either of these boxes can be positive or negative, so, having a, 'From' value as -3 and a, 'To' value as -5, the block will repeat 3 times. The, 'Range' feature also allows you to include an optional integer (large) variable to include as an **indexer**. The indexer variable will hold the current value of the loop's index. The loop's index is incremented by 1 on each iteration of the loop. So, if you have 1 in the, 'From' box and '5' in the, 'To' box, the indexer variable's value the first time through the loop will be 1. The index then increments by 1, so the second time through the indexer variable's value will be 2. The third time through it will be 3, then 4, then 5.
Spoiler alert - Advanced (if it wasn't enough already): Since the **indexer variable's value can also be altered to indicate the loop's index**, you can reset or, 'step' the index if you need to. If an indexer variable's value is altered, the loop's index will be set to match the indexer variable's value on the next iteration of the loop. Here are some examples. Let's say you have 1 in the, 'From' box and 10 in the, 'To' box. If you have a condition within your loop where you need to start the loop over, you can set the indexer variable's value to 1. On the next iteration of the loop, the index will now be 1 and the loop will act like it has started over. If you need to bail out of the loop (and don't want to use jumps/jump markers), you can set the value to 11 and the loop will exit before the next iteration starts (since the index will be greater than the value of 10 in the, 'To' box). If you need to, 'step' by a certain number, simply add that number to the indexer variable's value. So, if you have 2 in the, 'From' box and 10 in the, 'To' box and you want to just index by even numbers, simply add 2 to the indexer variable on each iteration. Hope I didn't just scare you off. Come visit everybody in the VoiceAttack User Forum and maybe we can clear all this up ;)

Note the, 'Indexer' box can also contain any number of tokens that can resolve to a variable name.

- Loop Start - Repeat Indefinitely (Infinite Loop)

This action indicates the beginning of a looping block that will loop continuously. The only way to stop this type of loop is with a **Loop Break** action (see below), or by stopping the command.

'Add a Loop End'

Use in conjunction with a, 'Loop Start' action to indicate the end of the block that requires looping. Command flow will go to the command action immediately following this action when the loop condition is not met.

'Add a Loop Break'

If a, ‘Loop Break’ action is encountered within a looping section, control flow is then moved to the end of the containing loop.

'Add a Loop Continue'

If a, ‘Loop Continue’ action is encountered within a looping section, control flow is then moved to the next looping iteration, or exits the loop if there are no more iterations.

'Add a Jump Marker'

This command action lets you place a marker within your command that will indicate a location that can be, 'jumped' to (using a 'Jump' command action).

Jump markers can be named whatever you want, but must be uniquely named within the command (if markers happen to be named the same in a prefix/suffix situation, the first marker will be the target of the jump). See, 'Add a Jump' (below) for more info.

'Add a Jump'

This action will instruct your running command to jump to another place within in your command’s actions. You'll notice that you can jump to three different places: To a marker, to the exit of the command and to the start of the command. If you choose to jump to a marker, you can choose an existing marker from the dropdown list, or just type in a name (in case you haven't created the marker yet, or, if the marker exists in a corresponding prefix or suffix command and is not available). Note that this input box also supports the use of tokens (see token reference near the end of this document).

If you choose to jump to the exit, the command will do just that... exit. Note that any sub-commands that are currently executing will continue to execute (this is not a kill switch for the command).

If you choose to jump to the start of the command, the command will continue processing from the start of the command.

Note that all jump types will work in prefix/suffix commands when they are executed together as a composite command.

'Exit Command'

This action allows you to simply exit the command. This will not stop any executing sub-commands, as this is just a simple exit (not a kill command). Note: This is just a more obvious implementation of what was previously only available as an option in a Jump (see above).

‘Get User Input’

An inline function or plugin makes sense when it comes to presenting a dialog to the user to get input. Also, a number of specific commands may be required to get a proper spoken response from a user. Sometimes you just want something basic to interact with the user, and that’s what the, ‘Get User Input’ set of features will do. The, ‘Get User Input’ screens consist of spoken response, text, integer, decimal and pick list (choice).

‘Get User Input – Wait for Spoken Response’ - This action will make the executing command wait until the user has spoken a response from a specific set of responses. The response is then placed in a text variable that can be examined with a conditional statement, so that the command can be instructed to do certain things. This action also provides a timeout that you can specify, so that if the response takes too long, the command can continue. There are two items that are required for this action to work. First, you must provide some type of response(s) in the, ‘**Responses**’ input box. This can be as simple as one word, (‘fire’), or, it can be a set of several words or phrases separated by semicolons (‘fire;fire weapons;fire at will’). Also, this input box will process dynamic phrases (‘fire[the;all;][weapons;lasers;squirrels]’) (see, ‘Dynamic command sections’ earlier in this document for help on that). This box will also render any combination of tokens to help broaden your response possibilities. Note that the maximum number of expected responses is limited to 500 items, and if more than 500 items are rendered via token, only the first 500 will be used (as you can tell, this is an advanced feature).

Second, you must indicate a text variable name in the, ‘**Text Variable**’ input box to hold the user’s spoken response. Note that this box also will resolve tokens into variable names. Optionally, you can specify a value in the, ‘**Timeout**’ input box. The timeout value indicates how long this action is to wait (in seconds) for a proper response before giving up and continuing. A value of zero indicates no timeout, which means that the action will wait for a response indefinitely. If the timeout period expires, the action will continue and the value placed in the indicated text variable will be, Not set. If you would like the command to continue no matter what the user says, check the, ‘**Continue on any Speech’** checkbox. If the user says something that is not in your provided response list, control will move from the action, but the value placed the text variable will be, ‘@@invalid’ (without the quotes).

Note: Again, if the user replies with a proper response, the text value of their response will be placed in the indicated text variable. This response will already be lower case for easy comparison in your command.

Note: The number of responses indicated in the, ‘Responses’ box must resolve to a maximum of 500 possible responses. This is an arbitrary limitation set to prevent the overloading of the speech engine.

‘Get User Input - Text’ - When this action is executed, the user will be presented with a simple dialog box that allows the user to type in whatever they would like. The user can click, ‘OK’ or, ‘Cancel’ to submit or not submit their information. This action will **require** you to indicate a text variable name in the, ‘Storage Variable’ field. This is the

text variable that will be populated with the result of whatever the user enters. If the user clicks, ‘OK’, the value in the variable will be what the user types in. If the user clicks, ‘Cancel’, the value in the variable will be, ‘Not set’.

The other values for this action are optional but are important for user presentation.

The, ‘Window Title’ option allows you to specify what will appear in the top bar of the displayed dialog. This can be a text variable name that contains the text to display. This can also be a literal value like, ‘Enter Some Text’, or can contain any combination of literal text and tokens. **NOTE:** Since this option can contain either a variable or literal text, if the variable’s value results in, ‘Not set’, the action assumes that the value entered is literal text. So, if you have a variable named, ‘myVariable’ and its value is not set, the window title will display, ‘myVariable’.

The, ‘Prompt Text’ option lets you specify some instructions just above where the user will type their input. This can be a text variable or literal text/tokens just like, ‘Window Title’ above. **Note**: Tokens such as, ‘{NEWLINE}’ can be handy here.

The, ‘Initial Value’ option lets you indicate a prepopulated value in the input box as a matter of convenience to the user. So, let’s say you are requesting info from the user, and the answer they are going to provide is most likely going to be, ‘Bananas’, you can indicate that here. ‘Bananas’ will appear in the input box and the user can choose to keep that value if they want to (and be spared from having to type it in). The same variable/text/token convention in the previous options apply here as well.

The, ‘Maximum Length’ option allows you to specify the maximum number of characters the user is going to type in. So, let’s say you are asking for some information that will only ever be 3 characters in length, you can specify that here. If you do not care about a maximum length, simply put a zero in this box.

The, ‘Require Input’ option lets you require that the user type something in. The, ‘OK’ button will not be enabled for the user as long as the input box is empty.

The, ‘Stay on Top’ option indicates that the input screen should be the top-most form when it is displayed.

‘Get User Input - Choice’ - When this action is executed, the user will be presented with a simple dialog box that allows the user to pick an item from a drop-down list of items. The user can click, ‘OK’ or, ‘Cancel’ to submit or not submit their selection. This action will **require** you to indicate a text variable name in the, ‘Storage Variable’ field. This is the text variable that will be populated with the result of whatever the user chooses. If the user clicks, ‘OK’, the value in the variable will be set to be the text of the item that the user selects. If the user clicks, ‘Cancel’, the value in the variable will be, ‘Not set’.

The other values for this action are optional but are important for user presentation.

The, ‘Window Title’ option allows you to specify what will appear in the top bar of the

displayed dialog. This can be a text variable name that contains the text to display. This can also be a literal value like, ‘Choose an Item’, or can contain any combination of literal text and tokens. **Note:** Since this option can contain either a variable or literal text, if the variable’s value results in, ‘Not set’, the action assumes that the value entered is literal text. So, if you have a variable named, ‘myVariable’ and its value is not set, the window title will display, ‘myVariable’.

The, ‘Prompt Text’ option lets you specify some instructions just above where the user will make their selection. This can be a text variable or literal text/tokens just like, ‘Window Title’ above. **Note**: Tokens such as, ‘{NEWLINE}’ can be handy here.

The, ‘Values’ option, although technically optional, must be provided if you actually want your user to have a selection to choose from. In order to indicate multiple items for your user to select from, you must separate your items by semicolons (;). For example, if you want to provide a pick list consisting of Apples, Oranges, Bananas and Pluto, the value entered into this box must look like this:
Apples;Oranges;Bananas;Pluto. The user will be presented with a pick list of those four items to choose from. The value in this box can also be a text variable or literal text/tokens just like the previous options.

The, ‘Selected Value’ option lets you indicate the selected item in the pick list when it is presented to the user. So, let’s say that you want the selected item to be, ‘Pluto’. Simply put, ‘Pluto’ in the box. When the pick list is displayed, ‘Pluto’ will be the item that is selected. **Note:** If the pick list does NOT contain the selected value, the selected value will be added to the pick list (and selected).

The, ‘Stay on Top’ option indicates that the input screen should be the top-most form when it is displayed.

‘Get User Input - Integer’ - When this action is executed, the user will be presented with a simple dialog box that allows the user to type in an integer value. The user can click, ‘OK’ or, ‘Cancel’ to submit or not submit their information. This action will **require** you to indicate an integer variable name in the, ‘Storage Variable’ field. This is the integer variable that will be populated with the result of whatever the user enters. If the user clicks, ‘OK’, the value in the variable will be what the user types in. If the user clicks, ‘Cancel’, the value in the variable will be, ‘Not set’.

The other values for this action are optional but are important for user presentation.

The, ‘Window Title’ option allows you to specify what will appear in the top bar of the displayed dialog. This can be a text variable name that contains the text to display. This can also be a literal value like, ‘Enter a Number’, or can contain any combination of literal text and tokens. **NOTE:** Since this option can contain either a variable or literal text, if the variable’s value results in, ‘Not set’, the action assumes that the value entered is literal text. So, if you have a variable named, ‘myVariable’ and its value is not set, the window title will display, ‘myVariable’.

The, ‘Prompt Text’ option lets you specify some instructions just above where the user

will type their input. This can be a text variable or literal text/tokens just like, ‘Window Title’ above. **Note**: Tokens such as, ‘{NEWLINE}’ can be handy here.

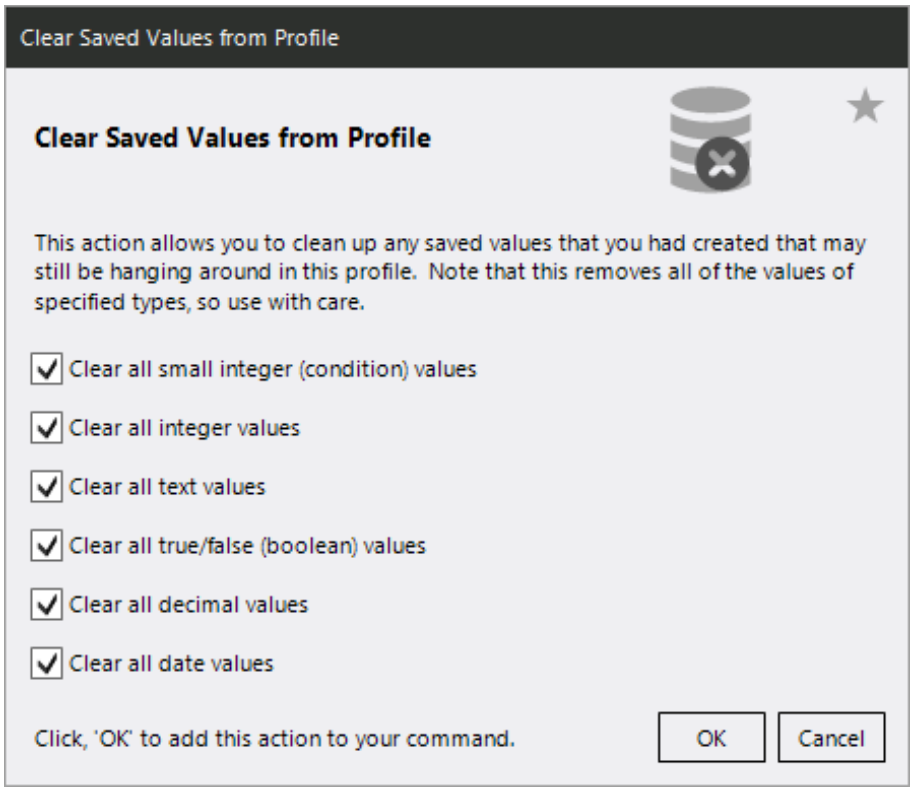
The, ‘Initial Value’ option lets you indicate a prepopulated value in the input box as a matter of convenience to the user. So, let’s say you are requesting info from the user, and the answer they are going to provide is most likely going to be, ‘55’, you can indicate that here. ‘55’ will appear in the input box and the user can choose to keep that value if they want to (and be spared from having to type it in). The value of this box can be a literal value (such as 55), or it can be a variable name that resolves to an integer or any combination of tokens that resolves to an integer value. If an integer value cannot be resolved, 0 will be used (or, whatever the minimum value is set to (see below)).

The, ‘Minimum’ and ‘Maximum’ options allow you to specify a range to constrain user input. So, if you know your user’s input is going to be between 1 and 10, you can put 1 in the, ‘Minimum’ box and, 10 in the ‘Maximum’ box. The OK button will not be enabled unless the user inputs a value in the range you specify. If you do not want to specify a range for either, ‘Minimum’ or ‘Maximum’ just leave either box blank. Note that the same variable/token conditions apply here as they do for the, ‘Initial Value’ option above.

The, ‘Stay on Top’ option indicates that the input screen should be the top-most form when it is displayed.

‘Get User Input - Decimal’ - This action behaves exactly the same as the ‘Get User Input - Integer’ action above, except that all the input is in decimal form and not integer.

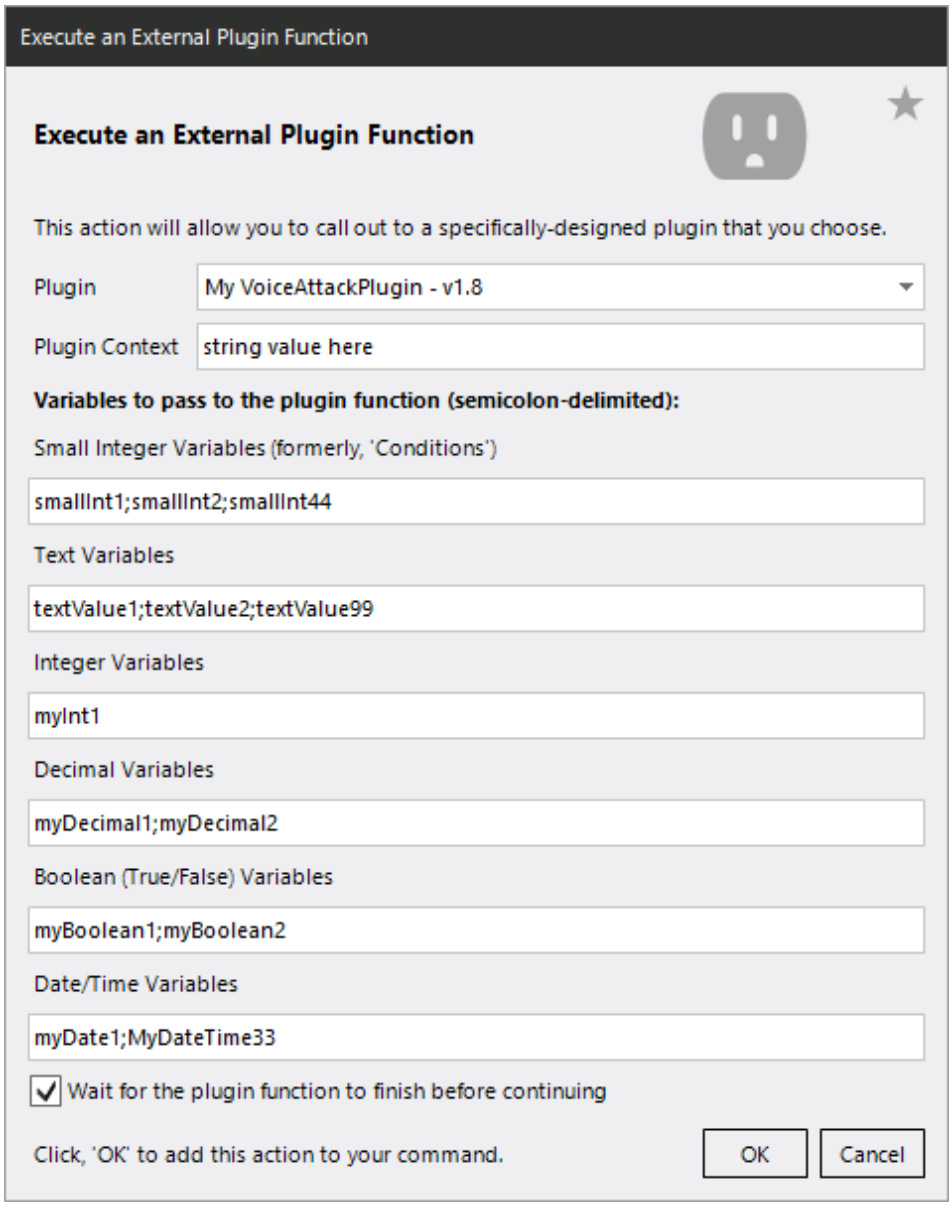
'Clear Saved Values from Profile'



This action will allow you to clear all saved variables of a given type from a profile. This is an easy way to initialize or clean up what has been persisted to disk. Simply select the data type(s) to clear using the check boxes provided.

Note: This will clear the variable values that are saved to the profile, but will NOT clear the resident variables from memory.

'Execute an External Plugin Function'



This command action is what you will use to invoke an external plugin function (see plugin section near the end of this document). To invoke a plugin, you must have enabled VoiceAttack plugin support (see, 'Options' page). If you have plugins installed, you can pick one to invoke from the dropdown list. To know how to interact with a plugin, you must see the documentation provided by the developer of your plugin. The plugin developer will explain what values need to go in which of the several boxes. Below is a short reference on what each control does.

Plugin Context - This is a string value that can be used to pass a simple value to the plugin. This can be anything you want, including any combination of replacement tokens.

Small Integer Variables (formerly referred to as, 'Conditions') - This is a semicolon-

delimited list of small integer (condition) variable names that you want to pass to the plugin. They can be small integers that already exist (see, 'Set a Small Integer (Condition) Variable' command action above), or new values that you want the plugin to fill (optional). The values can be modified in the plugin and returned to VoiceAttack for further processing.

Text Variables - This works exactly the same as small integers, except you are passing the text variables instead.

Integer Variables - This works exactly the same as small integers, except you are passing integer variables instead.

Decimal Variables - This works exactly the same as small integers, except you are passing decimal variables instead.

Boolean Variables - This works exactly the same as small integers, except you are passing Boolean variables instead.

Date/Time Variables - This works exactly the same as small integers, except you are passing date/time variables instead.

'Wait for the plugin function to finish before continuing' - This option allows you to have VoiceAttack wait until the plugin is finished so that you may have a chance to react to changes that have been made in the plugin. For instance, maybe the plugin goes out to the internet and retrieves data. The plugin packages up the data and returns it to VoiceAttack. If this option is checked, VoiceAttack's next action in sequence could do something with that data (like read it with text-to-speech, or invoke some other command based on a condition).

'Execute an Inline Function: C# or VB.net Code'

This will allow you to write C# or VB.net code that will be compiled and executed as a VoiceAttack action. Within this code, you have access to the VoiceAttack proxy object, ‘VA’, that is the same object used within the plugin framework so you can execute commands, get/set variables, parse tokens, etc. (see, ‘Plugins for the Truly Mad’/’Plugin Parameter Notes’ section later in this document for information on using this object). Since this is an advanced, specialized feature that requires a fair amount of detail, the discussion about this feature will be on the VoiceAttack user forums. For now, a light description will be provided here.

The first thing you will probably notice is the big code window right in the middle. This is where you will write your function(s). In there, you’ll see various, optional using/Imports statements that you would find in a typical new C# or VB.net forms project. You’ll notice that there is a **required** function, ‘main’ that needs to be present, as well as the class that encloses it (with **required** name of, ‘VAInline’). From, ‘main’, you can call your functions or instantiate your classes (or just put it all right in, ‘main’). As stated above, you will have access to a dynamic-typed VoiceAttack proxy object called, ‘VA’. When you create a new, ‘Inline Function’, you’ll see some commented-out examples showing how to use, ‘VA’. Note this is a very basic editor that will probably

evolve as time permits ;)

Above the code editor is the, '**Referenced Assemblies**' box. Within this box, you'll see an initial set of common referenced assemblies that are separated by semicolons. You can add or remove assemblies from this list as you need them for compiling your inline function (see notes on references below). Note that full paths to assemblies can be used (e.g., "C:\MyAssemblies\SomeAssembly.dll"), as well as a relative path to the VoiceAttack executable (e.g., "\Shared\Assemblies\SomeAssembly.dll"). This box will also accept tokens.

Some notes on references:
The assembly references that you indicate when you are compiling your inline function must be available to VoiceAttack when that code is actually executed. That is, all assemblies that are used by your code must be able to be found by VoiceAttack in order for your code to run. So, even though your inline function will compile (because your referenced assemblies are visible by the compiler using your explicit paths), you may receive an error when trying to run your code (either by running by clicking the, 'Test Run' button or when your function is called from within a command) as VoiceAttack is relying on standard search rules to locate your referenced assemblies (outlined below).

For inline functions that are only compiled on-the-fly (that is, NOT precompiled but compiled just before it is run), your referenced assemblies must reside in one of three places: The Global Assembly Cache (or, 'GAC'… which is where all the .Net framework assemblies reside. All the default assemblies in the reference box will typically be in the GAC), in the VoiceAttack installation root directory (where VoiceAttack.exe resides… kinda messy, tho) or in the Shared\Assemblies folder (this is inside the VoiceAttack installation folder and is created when VoiceAttack is installed). If your inline function is precompiled, your referenced assemblies can exist in any of the locations listed or can reside in the same directory as the inline function's compiled assembly. Again, your code may be able to compile successfully but you will receive an error if the code is run and VoiceAttack cannot locate your referenced assembly in any of the places listed above.

Down below the code window is a simple box for a description or name that will show up in the action list (instead of just, 'Inline C# function').

The, '**Wait for the inline function to finish before continuing**' option will cause the calling command to wait until the, 'main' function is finished before it resumes executing the next action. This is referred to as running the inline function *synchronously*. If this option is **not** checked, the inline command will be run *asynchronously,* and the command will resume execution of its next action immediately. Note: Running your inline function asynchronously will make it so that the code you write will run on in the background with no way to stop it with a command stop (by pressing the, 'stop all commands' button or by issuing a, 'stop commands' action). In order to help with this type of situation, the VoiceAttack proxy object ('VA') can be queried to find out if a command stop has been issued. Simply put a check in your code using the, '**Stopped**' property so that you can stop your function if it needs to be stopped with a command stop. Additionally, if you need to reset this flag for

whatever reason, the, '**ResetStopFlag()**' function can be used.

The, '**Retain Instance**' option will allow you to keep the instance of the class resident for subsequent calls. When an inline function is executed, an instance of the 'VAInline' class is created and then the, 'main' function is called. When this option is **not** selected, the instance of 'VAInline' is destroyed when, 'main' completes. The next time the inline function is called, a fresh, new instance is created and the cycle starts over. If this option is selected, the instance is not destroyed when, 'main' completes and any properties that are set within the instance are maintained. This is so you can maintain your information between subsequent calls.

The, 'Compile' button will attempt to compile the code that is in the code window with the assembly references listed. The box below the code window will show the status of the compile (errors, warnings or confirmation of a successful compilation).

The, 'Test Run' button will allow you to test run the, 'main' function. When you click, 'Test Run', the code is first compiled (just as if you clicked the, 'Compile' button) and then run if successful (0 errors). If the code finishes, you will see, 'Test Run Complete' in the status box. If your function runs on a while, you'll notice that the, 'Test Run' button has changed to, 'Stop'. You can stop your function by pressing the button again.

Even more advanced, in case you haven't had enough…

Normally, your *compiled* code in an Inline Function is not stored on disk. The function is completely kept in memory and only becomes compiled the first time it is run after VoiceAttack is launched. Sometimes, for whatever reason, you may not want to just have your code available to the end user, or, if your code requires a long compile time (probably not likely in this context, but you never know, right?) you may want to **precompile** your code into its very own file. You can do this by adding a file path to the, '**Build Output**' box and then clicking on the, '**Build**' button. The, 'Build' button works exactly like the, 'Compile' button, except that when the code is successfully compiled, the compiled, 'function' (also called an, 'assembly') will be written to the file location specified in the, 'Build Output' input box. The new file can then be referenced and executed by the, '**Execute an Inline Function: Precompiled**' action outlined below. Note that the, 'Build Output' and, 'Build' buttons are specifically for this purpose and are not used for anything else (Frankly, I didn't want to create an entirely separate set of screens just to do this one little bit that maybe two people out there *might* use… you'll have to forgive me ;)).

Note: Since there is (currently) no debugger, you will need to use the VA.WriteToLog() function (detailed later in this document) to write values out to the VoiceAttack log.

'Execute an Inline Function: Precompiled'

This will allow you to execute a previously-compiled inline function that you or somebody else create. This action goes along directly with the, 'Execute an Inline Function: C#/VB.net Code' action above. An option of the, 'Execute an Inline Function: C#/VB.net Code' action is the ability to compile your code to a specified file. The resulting, 'function' (also called an, 'assembly') can then be referenced from this action

and executed. To specify the file to use, just click on the '…' button to browse and select the file. You can then optionally provide a description for display purposes. The options, '**Wait for the inline function to finish before continuing**' and '**Retain Instance**' work exactly like they do in the, 'Execute an Inline Function: C#/VB.net Code' action listed above.

Note: When using a compiled function provided by somebody else, make sure that you trust the party from which you received the file. A malicious function can cause serious harm or cause security to be compromised in your system and/or network.

'Write Text to a File'

This action will allow you to write text to a file. The value to be written can contain literal text or tokens. Just indicate the text you want to write in the, 'Text' input box. You will then need to select a file to output your value to. Just click on the, '…' button to browse for a file, or just type in the full path in the, 'Output File' input box.

There are a couple of options that you can select. The first is whether or not you want to append the value to the target text file. If you choose the, 'Append text to file' option, the value will be written to the end of the target file. If the file does not exist, it will be created first. The next option is whether or not to overwrite an existing file. If the target file already exists and this option is selected, the target file will be completely overwritten. Make sure you use these options with absolute care, as you can erase or corrupt important data.

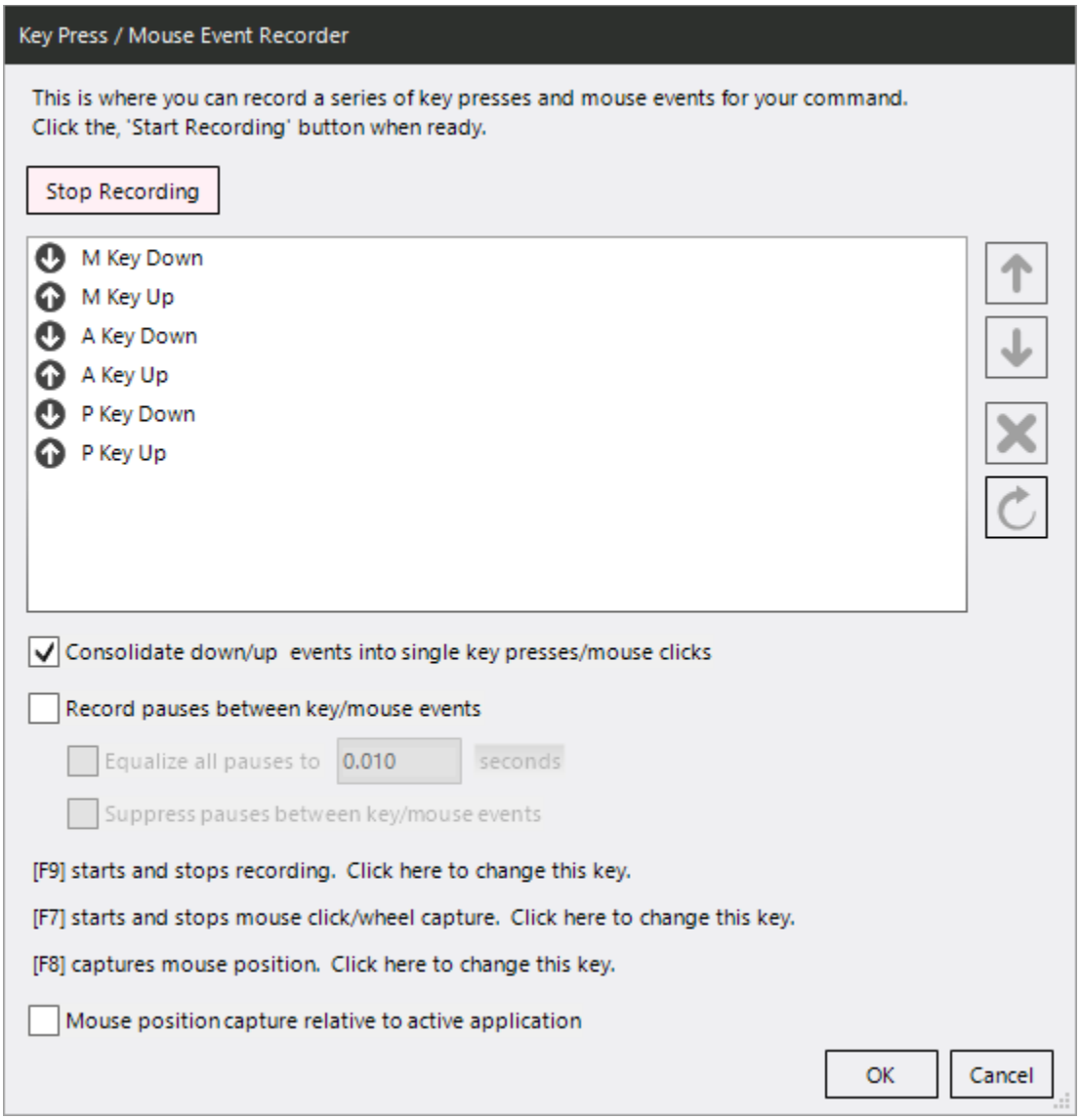
'Write a Value to the Event Log'

This action is very simple... just print some text to the VoiceAttack event log. Although not technically an, 'advanced' feature, this command action was created to assist in the development of commands that contain conditions and invoke plugins. The value that you output to the log can contain as many tokens as you need. You can also specify a color-coded dot if you want:)

'Add a Comment to the Action List'

This action is simply a comment that you can add to the action list for your own notation. It has no effect when executing commands. Note that this can be blank to add some spaces between actions (just to pretty things up a bit).

Key Press / Mouse Event Recorder Screen



This screen allows you to capture key presses and mouse events as you perform them. It will help you to create more lengthy macros quickly, as well as provide a better way to mimic human keyboard and mouse interaction (as required by a lot of games).

To start recording key press and mouse events, click on the, 'Start Recording' button. Click this button again to stop event recording. Note that you can press an assigned keyboard hotkey to start and stop recording, and that this hotkey can be changed by clicking on the appropriate link. In the illustration, ‘F10’ is assigned as the hotkey, and ‘F10’ can be pressed to toggle event recording on and off.

Once recording, to capture keyboard key presses, simply start typing. You’ll notice that each key down and key up is recorded in the event list, along with any pauses between each event. In this example, we are not recording the pauses that occur between key events. To record pauses, check the, 'Record pauses between key/mouse events' box. If you want all of the

pauses to be the same time value, check the, 'Equalize all pauses' box, then change value to what you would like the pauses to be. If your recording is very verbose (key X down, pause, key X up or mouse button X down, pause, mouse button X up), select the, 'Consolidate down/up events' box to consolidate multiple key or mouse button press events into a single command action (in the above example, M Key Down and M Key Up would become, 'Press and Release the M Key'). If you would like to omit pauses between the different key presses, simply select the, 'Suppress pauses between key/mouse events' option.
To include mouse clicks/scroll/tilt wheel events in your recording, you must first press the assigned hotkey. This is so that your mouse click recordings will start precisely when they should (and not when clicking away from VoiceAttack or moving windows around :)). To stop recording mouse clicks, simply press the assigned hotkey again. Note that the hotkey can be reassigned by clicking on the appropriate label.

You can capture the mouse position in your recording by pressing its own assigned hotkey as well. In the illustration, the ‘F8’ key is the assigned hotkey. Using the illustration as an example, pressing, ‘F8’ would capture the mouse position and display the position in the event list. Again, this hotkey can be reassigned by clicking on the appropriate label. Note that there is an option available when capturing the mouse position called, ‘**Mouse position capture relative to active application**’. When selected, this option instructs the recorder to capture the mouse location as it relates to the active, focused application. When this option is not selected, the recorder records the mouse location as it relates to the screen.

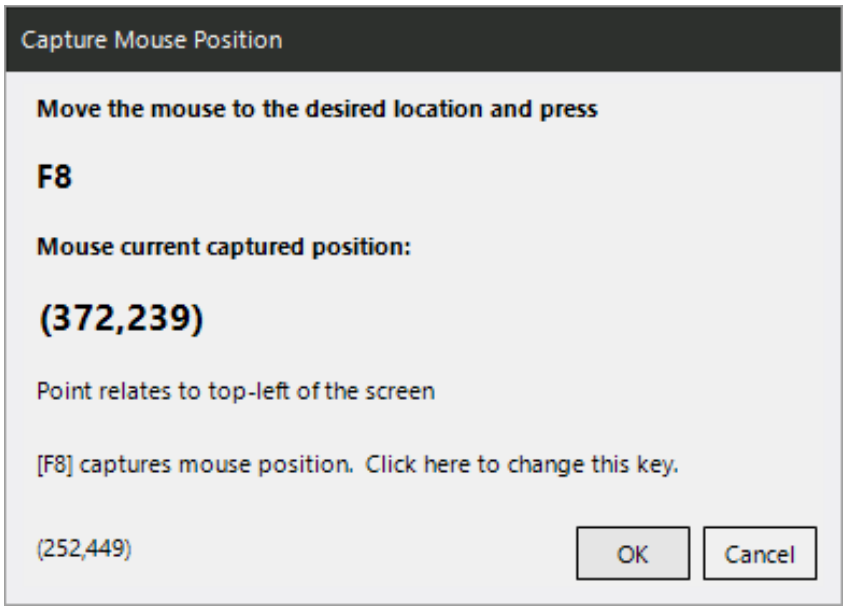
When you are satisfied with your recording, click the, 'OK' button to insert your recording into your command.

Mouse Move Screen

The Mouse Move screen allows you to add mouse move actions to your macros

Move to coordinates (X, Y)

Selecting this will allow you to move the mouse to a particular point on the screen. There are two ways of inputting the coordinates. First, you can manually enter the X and Y values in the input boxes, or you can click on the, 'Set Position' button to open up the 'Capture Mouse Position' helper screen:



To capture the mouse position, simply open your game or application and move your mouse to the location that you want and then press the designated hotkey / hotkey combination (in this example, the hotkey combination is Alt + F3). The coordinates of the mouse will then be displayed. Click, 'OK' to keep the coordinates.
Note: The hotkey can be reconfigured by clicking on the link provided.

There are several options that go along with the setting the mouse position. The first two are ‘Screen Coordinates' and 'Application Coordinates'. Choosing, 'Screen Coordinates' indicates that the mouse position is in relation to the entire screen (this includes multiple-monitor setups). So, if your coordinates are (100, 100), you'll notice your mouse moves close to the top-left edge of your monitor.
'Screen Coordinates' is good for a lot of cases (especially games) but if you are using windows to do tasks, this could be almost useless (since you move you move your windows around all the time). To make the mouse position relative to the current, active window, choose 'Application Coordinates'. If the coordinates are (100, 100), you'll notice that the cursor is placed near the top left corner of whatever window you are looking at. Note that the, 'Capture Mouse Location' helper respects this setting.

The other four options for setting the mouse position deal with how the position relates to a particular corner (either the corner of the screen or the corner of a window). The default is Top-left of screen/window (since this is the option you will probably be using

almost exclusively). The second-most used option will be the Bottom-right of screen/window (useful when windows are resizable… you know… so you can click that button that moves when your screen is resized). Your mouse coordinates will be in relation to the bottom-right of the screen or active window. Notice that the effective coordinates will be negative numbers, since the origin is the bottom-right corner. I hope this makes sense (lol). The 'Capture Mouse Location' helper respects this setting also.

Move to text / token-based coordinates

This option allows you to specify your X and Y coordinates as tokens so you can, 'programmatically' set your mouse positions. If your X and Y tokens resolve to values that can be interpreted as integers, the position will be used. Otherwise, no movement will occur (info will be displayed in the log). If the evaluated coordinate is outside the boundaries of the area, the boundary will still be used. See the section on Tokens later in this document to find out what's available to you. Note: all variable types can be expressed using tokens.

Move to a specific location

This feature will allow you to move the mouse to specified locations. The locations are the top-left, top-right, bottom-left, bottom-right, center, top, bottom, left and right of certain screens. Choosing top-left, top-right, bottom-left or bottom-right will move the cursor to the corners of the target. Choosing center moves the mouse cursor to the middle of the target. Choosing left, right, top or bottom will move the mouse cursor to that area of the target, related to the current location of the cursor. The targets are:
‘Application’ - the executing command’s active target.
‘Primary Screen’ - the primary screen as indicated by Windows.
‘Cursor Screen’ - the screen where the cursor is located.
‘Active Window Screen’ - the screen where the active window is located (as indicated by the top-left corner of the screen).
‘All Screens’ - all active screens which constitute a, ‘virtual screen’.

Save Current Position

Choosing this option will make VoiceAttack remember the current mouse cursor position in your macro. This, 'remembered' position can be recalled with the 'Recall mouse cursor to saved location' option.

Recall mouse cursor to saved position

Adding this action to your command will move the mouse cursor to the last remembered position (after 'Save current position' is issued). **Note:** If no position is saved, the mouse position will not change. Additionally, when the 'Recall mouse cursor to saved location' action is issued, the remembered value is cleared.

Move Mouse Left / Right / Up / Down (Adjust the Mouse Cursor Location)

These actions will allow you to move your mouse a specified number of pixels. You can choose to move your mouse left/right and up/down. Simply select the radio button to select the direction and type the number of units to move in the box provided. The input boxes for each direction will allow you to input numbers (like 100), tokens (like, ‘{INT:myVariable}’ (see the VoiceAttack token reference later in this document for information on tokens)) and long integer variables (like ‘myVariable’). VoiceAttack will

attempt to resolve your numbers, tokens or variables into a value and move the mouse accordingly. If the value provided cannot be resolved into an integer, the value will be resolved as zero and the mouse will not be moved in that particular direction.

The option, '**Move from cursor position**' will make your mouse movement relative to the current location of your mouse cursor. The option, '**Move from last-moved position**' will make the mouse move from the last place that VoiceAttack had set it (note that if this position has not been set, the current mouse cursor location will be used).

Some 3D games require a certain type of input to work properly. The option labeled, '**Move using relative data**' is provided to assist with moving the mouse in, '3D space'. Try checking this box if your ship or your FPS character doesn't respond to mouse movements.

A fun feature added to the movement of the mouse is the ability to do some simple mouse animation. This was added to help with keeping track of the mouse, versus having it just move to another part of the screen instantly. To animate your mouse cursor movement, simply check the option labeled, '**Animate movement**'. There are some options that go along with animating the mouse that are required. First, you need to choose whether or not your movement will ease in to its destination or just move consistently. To make your mouse ease into its destination, select, '**Ease movement**', otherwise, uncheck the box. The next two parts go hand-in-hand: 'Timing' and, 'Steps'. Steps is the maximum number of steps to take when animating your mouse, and Timing is a base time for the movement. Increasing the timing makes the mouse cursor take longer to reach its final destination. Increasing the steps increases how smooth the mouse cursor seems to move. There are catches, however. Timing is based on seconds, but the end result will usually not be exact (especially if you add a lot of steps). The more steps you add, the more the base time is going to expand. On the flipside of this is if you use the, 'Ease movement' option, where the mouse cursor may reach its destination faster. You will want to play around the settings to get it just right for you. Easing the movement with timing of 1.00 and 60 steps works great here, but it may not with your system.

Mouse Click Screen

The mouse click screen allows you to add various click actions to your macros, including scrolling the mouse wheel.

Click Left / Right / Middle / Back / Forward button

These actions will make the mouse click at the current point on the screen using the selected mouse button.

Double-click Left / Right / Middle / Back / Forward button

These actions will make the mouse double-click at the current point on the screen using the selected mouse button.

Mouse Click Duration

Available only for click and double-click actions, this value is the amount of time in seconds between when the mouse button is pressed and when it is released. For most Windows applications, this value can be zero. For games, however, a value of zero is usually too fast to be detected reliably. The default value is 0.1 second, which may need to be adjusted for your application.

Left / Right / Middle / Back / Forward button down

These actions will make the mouse press down only at the current point on the screen using the selected button. This effectively simulates, 'press-and-hold', and, **you will need to add a subsequent release**.

Release Left / Right / Middle / Back / Forward button

These actions will make the mouse release the selected button. Use this as a follow-up to a previous mouse button down action.

Toggle Left / Right / Middle / Back / Forward button

These actions will make the mouse press down if it is up, or release if it is down (see above for more info).

Scroll wheel forward / backward

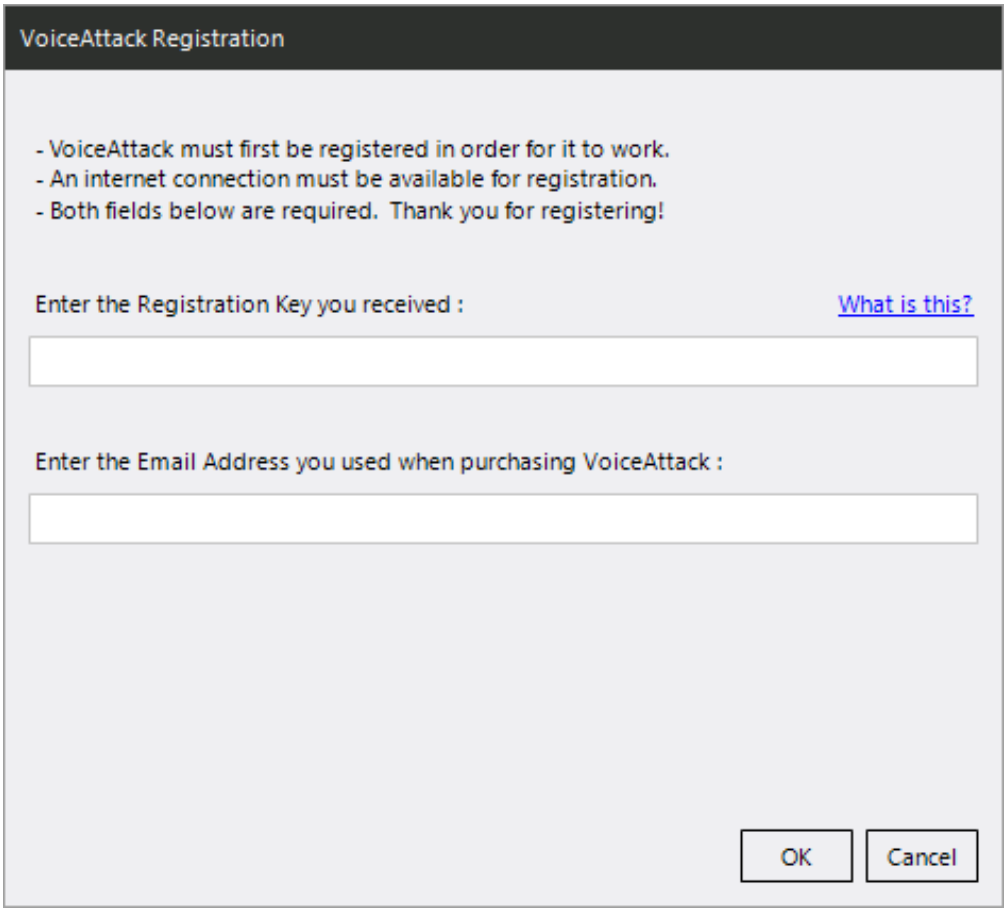
These actions will allow you to scroll the mouse wheel forward or backward by however many number of 'wheel clicks' that you choose.

Note: To simulate what an actual mouse does when clicking, any mouse down action will now set the command's process target to be the application of the window that is located under the mouse. This will be for the remainder of the executing command. In older versions of VoiceAttack, the active window or specified process target always received the command's mouse messages for input. This would cause problems unless you knew a lot about VoiceAttack. Not good for something that should be so simple, right?
In order to turn this feature off, select the option, 'Bypass Mouse Targeting' on the System tab on the Options screen.

Registration Screen

If you are using the limited trial version of VoiceAttack and are still within the trial period, you will see the splash screen below each time you run VoiceAttack. In the bottom-left corner, there will be an indication of how much time is left to use the trial.

If you have run out of time, or, you have clicked the 'Options' button on the main screen (See 'Options Screen') and then clicked the 'Registration' button, you will be presented with the registration screen for VoiceAttack:



If you have purchased a registration key from VoiceAttack.com, this is the place to put it in. Simply type or paste in your registration key into the box provided. Additionally, you will need to put in the very same email address that you used when you purchased the VoiceAttack registration key. Click the, 'OK' button to start the validation process. This should only take seconds depending on your Internet connection and firewall settings.

Upon successful validation, the registration screen changes to display only what you have just entered, just for your records. If validation is not successful (for various reasons), you can hit cancel to continue using the VoiceAttack trial until time runs out.

Options Screen

The, Options screen allows you to configure your VoiceAttack application. The Options screen and it's supporting dialogs are outlined below. Note that the Options screen now has three tabs with various settings on each tab.

General tab

Below are the items you'll find on the, 'General' tab of the Options screen.

Registration Button
Click this to bring up the registration screen (See 'Registration Screen').

Check for Updates Button
Click this button to have VoiceAttack check if an update is available for download (internet connection is required, of course). An additional option for this feature is 'Include Beta Versions', which will allow you to also check for VoiceAttack pre-release versions (if you like to keep up with the latest VoiceAttack builds).

Reset Defaults Button
Click this button if you want VoiceAttack to return all of the settings back to what they were when it was first installed. The screen presented also contains an option to remove all registration data. If the option is selected, the registration data will also be reset. **Note**: Profile data is not reset.

System Info Button
Clicking this button will allow you to display different information about VoiceAttack, your system and DirectX.

Launch with Windows Start
Checking this option will make VoiceAttack launch when Windows is started. This is a per-user setting. **Note**: If VoiceAttack is running as an administrator, Windows may prevent it from launching at startup.

Minimize to System Tray
When selected, a VoiceAttack icon is placed in the System Tray. When unselected, VoiceAttack minimizes to the task bar.

Check for Updates on Startup
Selecting this option will make VoiceAttack check for updates each time it is started. You will only get a message if there is actually an update available.

Start Minimized
Select this option to make VoiceAttack start up in a minimized window state.

Show Tips at Startup
When this is selected, the VoiceAttack Tips screen will be displayed each time VoiceAttack is launched.

Close Button Minimize Only

When this option is checked, clicking the, ‘close’ button on the main screen will just minimize VoiceAttack instead of closing it. Closing VoiceAttack can then be done from the task bar or the system tray (depending on if, ‘Minimize to System Tray’ is enabled or not).

Disable F1 Help

Select this option if you would like to turn off VoiceAttack’s reaction to the standard F1 help request feature. The help documentation can still be accessed via the system menu on the main screen (the icon in the top-left corner).

Show Control Tips

Tool tips pop up to give descriptions of most items. Deselect this box to turn them off. Default value is 'selected'.

Enable Auto Profile Switching

This turns the VoiceAttack automatic profile switching on and off for all profiles that are enabled. See the Profile Options page for more details on automatic profile switching.

Enable Plugin Support

This will turn on plugin support in VoiceAttack. Plugins are external pieces of code that can be custom-built to work with VoiceAttack. See the section entitled, 'Plugins' near the end of this document for more information. Note that this is an advanced feature of VoiceAttack, and you will receive a warning box indicating the dangers of enabling this feature.

Plugin Manager Button

When plugins are enabled, you can access the, ‘Plugin Manager’ screen. This basic screen will allow you to enable or disable specific plugins. Note that enabling or disabling plugins will require a restart of VoiceAttack to become effective.

Load Profile on Startup

This option will allow you to select a profile that gets loaded when VoiceAttack is launched. Setting this option to, ‘None’ will revert back to the default behavior, which is loading the last-used profile on startup.

Global Profiles

This option will allow you to reference, or, ‘include’ the commands from any or all of your other profiles with any current, active profile. This way, you can create common profiles filled with commands that can be shared among all profiles. The profiles that you include can be arranged in priority, from highest to lowest. When duplicate-named commands are encountered, **the command in the profile with the higher priority will be retained**. For instance, let’s say you have two profiles that you want to include: Profile A and Profile B. Profile A has been given a higher priority than Profile B (it’s higher up on the list). Both profiles have a command called, ‘Fire Weapons’. When you issue the command, ‘Fire Weapons’, the command from Profile A will be used, since Profile A has a higher priority.

Note: The profiles indicated in this set are lower priority than the, ‘Include commands from other profiles’ option on the Profile Options screen (See, ‘Include commands from other profiles’ on the Profile Options screen). Also, the current, active profile will always have the highest priority.

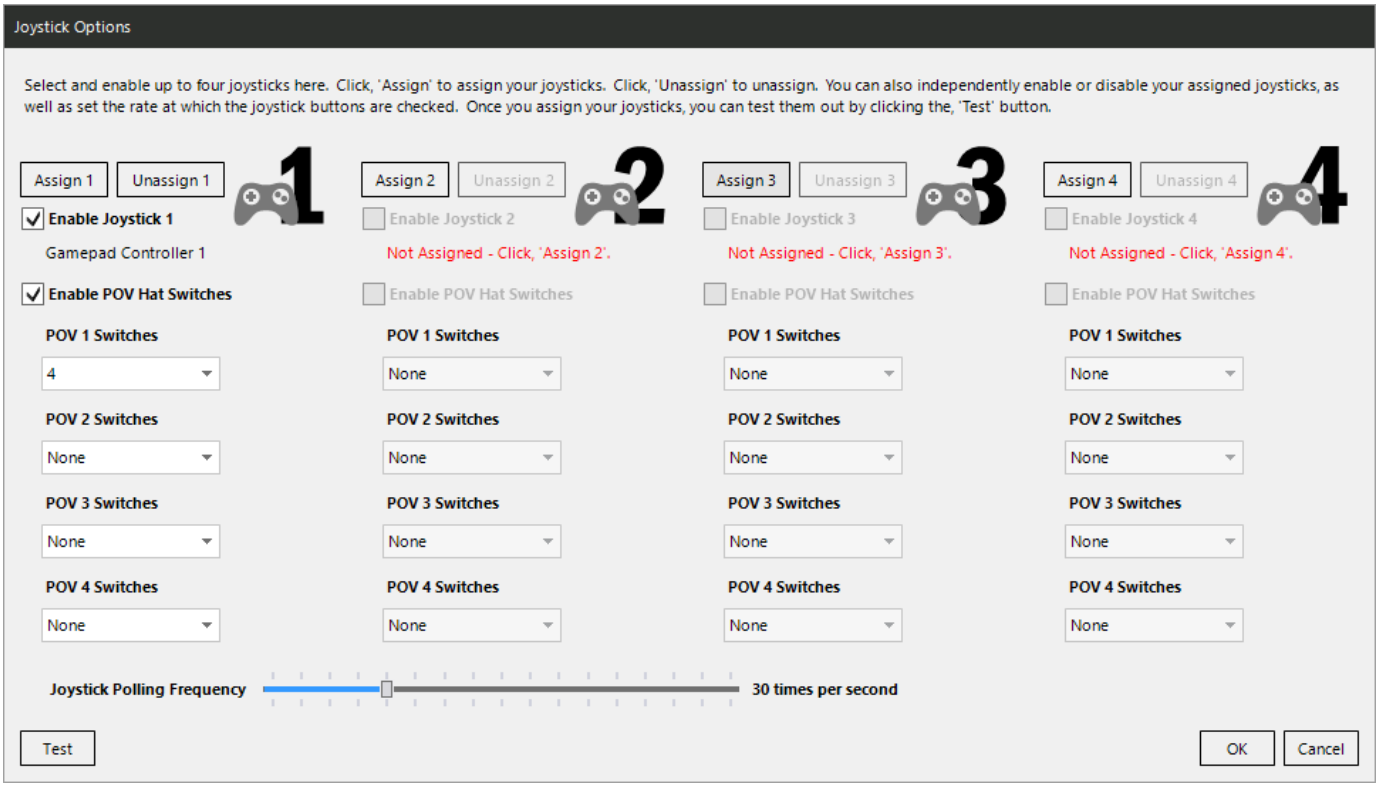
To edit the list of included profiles, click on the, ‘…’ button to the right. This will bring up the, ‘Include Profile Commands’ screen. Use the controls on the right side of the screen to add, arrange and delete included profiles. Click, ‘OK’ when you are finished.

Keyboard Display Layout

When using the Key Press screen, this option allows you to change what is displayed for selected keys. For instance, key code 51 maps to the, '3' key. If you are in the US, '3 #' is displayed. If you are in the UK, '3 £' is displayed. If your layout is not yet supported, VoiceAttack will use the US layout (which is what VoiceAttack had been using up to v1.5.3). For most users, this value will not need to be changed. If you would like to force the display to another layout, just drop down the list to select from the ones available (more to come).

Joystick Options

This is where you will assign and enable joysticks for use within VoiceAttack. Joysticks can be used to carry out various tasks, such as executing commands as well as turning VoiceAttack's listening on and off. Clicking on this button will take you to the Joystick Options screen:



From this screen, you will be able to assign up to four joysticks for use in VoiceAttack. You will notice when you first open this screen, you will have no assigned joysticks. To assign a joystick to use within VoiceAttack, click the, ‘Assign’ button next to

joystick slot 1, 2, 3 or 4. You will then be asked to select a joystick to assign from any currently-enabled devices reporting to be joysticks on your PC (**Note:** Controllers identifying as gamepads will have an extra, selectable option labeled as, ‘Gamepad Controller 1-4’. This extra selection is in place to assist those that are having compatibility difficulties using certain wireless controllers with VoiceAttack (weird, right?). If you are not having trouble with your gamepad controller, make sure to select the specifically-named option of your controller for maximum compatibility). Select the joystick that you want to assign and then you will be returned to the Joystick Options screen. Now that you have a joystick assigned, you can enable it or disable it by using the checkbox labeled, 'Enable Joystick (1,2, 3 or 4)'. To clear a joystick assignment, just click the, ‘Unassign’ button next to the appropriate assignment. You'll notice that you can also change the rate at which VoiceAttack checks the state of your joysticks. This value is 30 times per second by default, but you can raise or lower that rate as you see fit.

In the big middle of everything is where you can set up your POV (hat) controllers to act like simple switches. Your POV can become a switch that acts like 1, 2, 4 or 8 buttons. For instance, if you select 4, pushing forward, right, back or left on your POV will make VoiceAttack treat each position as POV button 1, 2, 3 and 4 respectively. If you select option1, VoiceAttack treats any direction as POV button 1. Selecting 2 (Up/Down), if you push up, that represents POV button 1. Pulling back represents POV button 2, and so on. VoiceAttack can support up to four POV controllers on each joystick if you've got 'em. In the bottom-left portion of the screen, you will see the, 'Test' button. If your joysticks are assigned and enabled, you can click on this button to try them out.

Sounds Folder

Part of the advanced features of VoiceAttack, the VoiceAttack Sounds folder makes it easy to have a central place to keep and maintain your VoiceAttack sound effects files. This folder can be located anywhere on your system and can be accessed via the {VA_SOUNDS} token (see tokens near the end of this document). Where this comes in handy is that it allows you to have kind of a virtual directory that you can export with your profile and share. Let's say you have a collection of sounds in C:\VoiceAttack\Sounds\VeryCoolSoundPack. Inside this sound pack folder, you have a sound file called, 'detonate.wav'. You can set the Sounds folder to C:\VoiceAttack\Sounds, and access the sound file in the sound pack directory like this {VA_SOUNDS}\VeryCoolSoundPack\detonate.wav. If you export this in a command in your profile, the token goes with it, so your friends do not have to have the same file structure as you do. They only need to have the Sounds folder set up with the, 'VeryCoolSoundPack' folder located appropriately.

Apps Folder

Just like the Sounds folder above, the Apps folder makes it easy to have a central place to keep your apps (.exe files) and plugins (.dll files). This is also a special folder for VoiceAttack in regards to plugins. VoiceAttack will only look in the subfolders of this folder to look for plugins (see section on Plugins near the end of this document).

Recognition Tab

Speech Engine

This will allow you to choose a SAPI-compliant engine. You may never need to change this. It was fun for us to mess with, but, 99.99999 percent of users will just need to leave this set to 'System Default'. More on this at a later time.

Recognized Speech Delay

This is the amount of time that VoiceAttack's speech engine waits to perform the actions of a command after it understands a phrase and detects a silence. Play around with this number. If you have phrases that are similar and lengthy, a higher value might be needed. If your phrases are short and different, go with a lower value. The value range is 0 – 10000 (10 seconds). The default value is 0 (it is recommended that you start at zero and work your way up if are having trouble).

A simple example of a reason to increase the value of 'Recognized Speech Delay' would be if you had two commands. The first one is, ‘pet attack' and the second one is, 'pet attack plus ten'. Depending on *your* speaking ability, VoiceAttack may execute, 'pet attack' if the value is set too low. Increasing the value causes VoiceAttack to wait before execution, so you can finish articulating your command.

Unrecognized Speech Delay

This is the amount of time that VoiceAttack's speech engine waits to reject a speech stream that it does not understand and then detects a silence. The value range is from 0 to 10000 (10 seconds). The higher the value, the longer VoiceAttack will take to reject the speech and move on.

An example of a reason to increase the, 'Unrecognized Speech Delay' value would be in a situation where you have a two-word command like, 'pet attack' (also, let's consider that you do not have a command called, 'pet'). Depending on *your* speaking ability, you may only be able to get out, 'pet' (which is unrecognized). Increasing the value causes VoiceAttack to not immediately reject the command, thereby allowing you to finish speaking your command.

Note: This setting is useful for those that leave VoiceAttack's listening on all of the time. It adjusts the delay after one finishes talking over their headset before being able to issue a command.

Command Weight

This value is the relative weight of the commands in your profile versus everything else you say. The higher the number, the more likely VoiceAttack will make a, 'best guess' at what you say. For example, when this value is at maximum (100), your commands have full weight. That means that basically **everything** you say will be interpreted into a command. If you say, 'flag', and, you have no command called, 'flag', but, the closest match is, 'bag', VoiceAttack will execute the command named, 'bag'. Note that a high value in this option will probably not be desirable when VoiceAttack's listening is on all of the time (especially if you carry on conversations). However, a high value may be beneficial when using the push-to-talk feature. Play around with this number to get the right balance for your speaking style.

Minimum Confidence Level

When the Windows speech engine recognizes a phrase, it provides a confidence rating of just how accurate it thinks it was at doing its job. VoiceAttack will allow you to filter out anything that the speech engine recognizes but does not meet a minimum rating. You can set this value here (from 0 to 100). The higher the number, the more selective VoiceAttack will be about executing commands. Note that this value can be overridden at both the profile level (on the Profile Options screen) as well as for each individual command (on the Command Add/Edit screen). Any phrase that is recognized but rejected because it falls below the minimum value will show up in the log.

Select the, 'Show confidence level' option to show the speech engine's confidence level in the log for each recognized phrase.

Min Unrecognized Confidence Level

Setting this value allows you to filter the, 'Unrecognized' log items a bit. The higher the value, the more likely your, 'unrecognized' log items will be filtered out. This will help cut down on log chattiness in certain noisy situations.

Disable Adaptive Recognition

The speech engine used by VoiceAttack is constantly learning from what it, 'hears'. When environments are noisy, the speech engine may become somewhat unresponsive. Although selecting this option is not recommended, it may be very helpful if you are using VoiceAttack in a noisy place.

Disable Acoustic Echo Cancellation

This will disable the acoustic echo cancellation on your pc in an attempt to increase recognition reliability. Note that this is a system-wide change and can affect other applications that may depend on this.

Reject Pending Speech

When you turn on and off, 'listening' in VoiceAttack, you can choose how to handle what happens if you are speaking at the same time you are toggling the, ‘listening’ state. When this option is **not checked** and you turn on or off, 'listening' at the same time you are speaking your command, VoiceAttack will continue to pick up what you are saying without cancelling it. That is, VoiceAttack to try to be a little more forgiving if you start or stop, ‘listening’ too early or too late. For instance, if you have a relatively long command phrase that you are speaking and you turn off, ‘listening’ right before you finish speaking or right before the command is fully recognized, your spoken phrase will continue to be recognized. Also, if you start speaking your phrase and you are a little bit late turning on, ‘listening’, your phrase will still be picked up.

When, ‘Reject Pending Speech’ is selected, you will have a more exact and expected control over when commands are picked up, as **‘listening’ must be turned on** for the entire duration of your spoken phrase - otherwise it will be rejected. For example, if you have a relatively long phrase and you turn off, ‘listening’ at any time while speaking, your entire phrase will be rejected. If you are speaking a command phrase and then turn, ‘listening’ on after you have started, your command phrase will also not be picked up. As you can see, although this option provides more control, you lose

somewhat of a buffer for human error. It's all about playing style, really ;)

Repeat Command Phrases

When, 'When I say' phrases get long they are long, it's kind of a chore to repeat them over and over again. It would be easier to be able to just say, 'repeat' or, 'repeat that' and have the last spoken command be executed again. This option allows you to specify, 'repeat' phrases that you can speak to execute (repeat) the last spoken command. If you want to simply say, 'repeat', just put the word, 'repeat' in the input box. Note that this is a semicolon-delimited list, so, if you also wanted to say, 'repeat that', just enter, 'repeat;repeat that' in the box (without quotes). Note also that this only repeats the last spoken command, and not commands executed by other means.

Recognition prefix exclusions

Sometimes when we speak commands there are ambient noises. You might breathe or make a, 'pop' noise. These noises are sometimes interpreted as words by the speech engine. For instance, if you have a command called, 'power to shields', you might see it come up as, 'Unrecognized: **if** power to shields'. This is because the speech engine interpreted the, 'if' from some kind of noise it picked up (mostly breathing). Items that you add to this semicolon-delimited list will be filtered out from the beginning of unrecognized phrases and reprocess those phrases to see if they are actually recognized. Adding, 'if' to the list will filter out the 'if' at the beginning of, '**if** power to shields' and recheck to see if, 'power to shields' is acceptable (which it will be in this case). Your command will then be recognized. Note that the default is, 'if;but;the' (without quotes). That means that all three of these words will be filtered from the beginning of all unrecognized commands. You will probably need to change these if you are not using the English speech engine ;)

For a long time, VoiceAttack has filtered out some words ('if' and 'but'). This doesn't work well with non-English speakers and they are hard-coded (of course). This option lets you pick what words to use. This is a semicolon-delimited list of values (the default is, 'if;but;the').

Windows Speech Recording Device

This is the recording device (microphone) that is selected by Windows for speech recognition. When VoiceAttack initializes the speech engine that it is using, this will be the device that is used. If you swap out microphones a lot, you might want to set this value to a device that you always use for speech recognition. If you only have one device, you would probably just want to leave this as, 'Default'. Once you have selected the device that you would like to use, you can set its volume from the volume slider to the right. Note that changing these values will change Windows' very own settings, so all applications that depend on these settings will then use the selected device. Also note that the settings are not saved unless you click, 'OK'.

Disable Speech Recognition

This option will allow you to turn off VoiceAttack's speech engine facilities. No check for a speech engine will be made at start up so you can execute VoiceAttack's commands using keyboard shortcuts, mouse buttons or joystick buttons. When this option is changed, VoiceAttack will need to be restarted before the change goes into effect (see also the, '-nospeech' command line option).

Utilities Button

This button contains shortcuts to common Windows applications regarding the speech engine, such as, ‘Speech Control Panel’, ‘Speech Engine Training’, ‘Add/Remove Dictionary Words’, ‘Microphone Setup’ and, if you are using later builds of Windows 10, access to the new, ‘Sound Settings’ app.

Audio Tab

Audio Output Type

This option has three selections, each with their own characteristics. You will want to choose the option that best suits your needs and/or your pc’s needs. Each is explained below.

Legacy Audio - This is the oldest selection, going all the way back to the beginning of VoiceAttack. This option is the most limited, but also great if your system doesn’t support the other options and you really need to hear sounds. Playing audio with this option limits you to .wav files, and also does not allow you to do other things like set the volume, start position, or wait for the audio to complete.

Windows Media Components - This is the second-oldest selection in VoiceAttack. It will allow you to set the volume of your sound and other options. The largest benefit of this method of playback is that **very** robust, in that it will seemingly play anything you throw at it sample-wise as long as you have a codec it can use. The drawback is that it **requires** Windows Media Player 10 or later to be installed on your system, since VoiceAttack will share the components that come along with it. You also cannot choose the audio output channel or pan the volume from left to right (not a big deal for most).

Integrated Components - This is the newest option in VoiceAttack (as well as the default selection on new installations). This selection affords you the most options when playing audio, such as volume, panning, output channel selection, etc. Another benefit is that Windows Media Player is not required to be installed. The only drawback is that this option will have a hard time playing back some audio files that have variable bitrates.

Notification Sounds

This turns on/off notification alert sounds within VoiceAttack (such as the sounds you hear when VoiceAttack starts and stops listening). Check the box to enable notification sounds, uncheck to disable. This option is enabled by default.

Fade Stopped Audio

When the playback of audio files takes place in VoiceAttack, often that audio is interrupted by a user action or command action. Stopping audio is usually abrupt and not very appealing. This option will attempt to fade out audio that is stopped and hopefully make your ears happy. Check the box to enable this feature, uncheck to disable. This option is enabled by default. Note that this feature does nothing if, ‘Legacy Audio’ is selected as your output type.

Override Default Playback Device

If you have, ‘Integrated Components’ selected as your audio output type (above), you’ll be able to select the output channel that your audio files play through (like desktop speakers, headset, Bluetooth device, etc.). When, ‘Default’ is selected as the channel, VoiceAttack simply plays the audio through the default playback channel as currently indicated by Windows. Choosing a device from this option will allow you to override the Windows default device with the one selected.

Override Default Text-to-Speech Device

This works exactly like, ‘Override Default Playback Device’ above, except this option controls the default channel that VoiceAttack’s text-to-speech audio is played through.

Stop Commands, Feature On and Feature Off Sound

For a little bit of added fun (and to possibly aid in immersion), you can set the sounds of the three base VoiceAttack sounds from a predefined set. Setting the, ‘Stop Commands Sound’ will change the sound when you stop a command. Setting the, ‘Feature On Sound’ and ‘Feature Off Sound’ options will change the on/off sounds in VoiceAttack (listening on/off, hotkeys on/off, etc.). Use, ‘Default’ for any of these to keep it like it’s always been ;) See the, ‘For fun… Maybe’ section at the end of this document for information on how to use your own sounds.

Audio Cache Size

When VoiceAttack reads an audio file from disk, it holds on to the file in memory so that repeated play of the same audio file does not have to be read from the disk over and over again. This option allows you to decide how much memory (in megabytes) that can be used by VoiceAttack to store these files. The maximum value is 256 megabytes. If you do not want to cache any files, set this value to zero. **Note:** Audio caching is only available if, ‘Integrated Components’ is selected as the audio output type.

Sound File Volume Offset

This is the offset value for the volume of all sound files that are played using the, 'Play a Sound' command action. Think of this as a main volume control for all of your sounds. **Note**: This feature only works if you are not using Legacy Audio.

TTS Volume Offset

This is the offset value for the volume of all Text-To-Speech playback using the, 'Say Something' command action. Think of this as a main volume control for Text-To-Speech in VoiceAttack.

Set Windows Default Multimedia Audio Playback Device on Startup

This is kind of a multi-function feature that allows you to set the Windows default audio playback device (like speakers or headphones) when VoiceAttack starts. Also, the, ‘Change Now’ button is provided as a convenience so you can change the audio device immediately.

VoiceAttack uses Windows Media components to do its thing, and since these components only work with the default playback device, this feature may save you a step (or several) while using VoiceAttack. To use this feature, simply select the device

you would like to use (the list only shows your currently-available devices). If you want to switch to the selected device on startup of VoiceAttack, just check the option box. If you want to change the device immediately, just click the, 'Change Now' button. There are also some command line options for controlling the default playback device. See, '-output' in the command line options section later in this document.

Note: This feature is only available for Windows 7 and later.

Note: If the device's underlying identifier is changed (driver update, Windows update, crash, etc.), VoiceAttack will attempt to resolve the device by its last-known device name. If you see a log message containing, 'Resolved by device name', VoiceAttack has successfully made the change, but you may want to update this setting.

WARNING: This is not a VoiceAttack setting, rather a Windows device setting and can (and probably will) cause other applications that depend on the changed devices to appear to malfunction. It's not THAT big of a deal, but it will definitely throw you off when your Skype or TeamSpeak is not working how you left them.

Set Windows Default Communications Audio Recording Device on Startup

This works exactly like the option above but for the default communications device. Everything applies, except you'll be looking for, '-outputcomms' in the command line options section.

Set Windows Default Multimedia Audio Recording Device on Startup

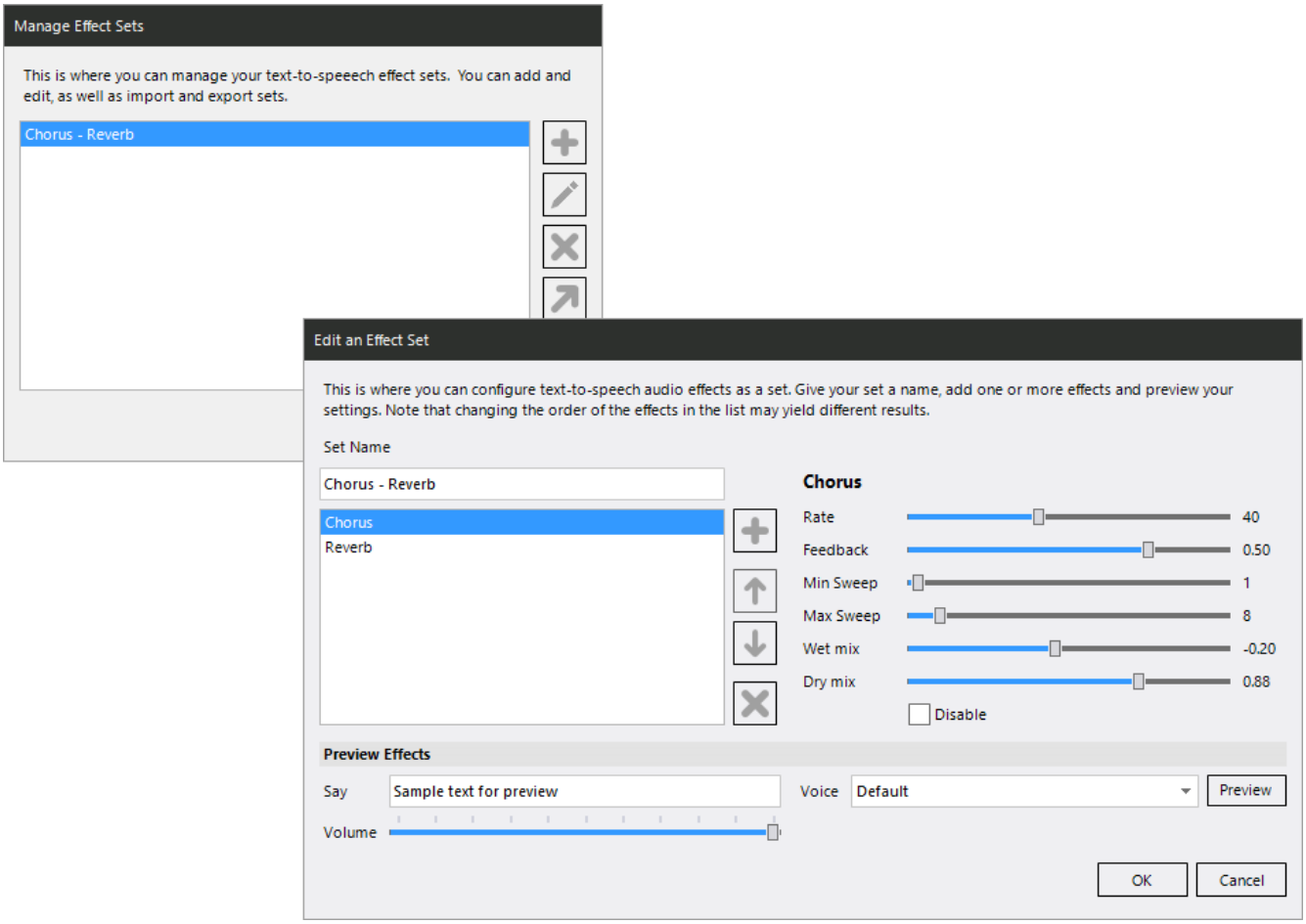
This works exactly like the options above but for the default recording device (table mic, headset mic, webcam mic, etc.). Everything applies, except you'll be looking for, '-input' in the command line options section.

Set Windows Default Communications Audio Recording Device on Startup

This works exactly like the option above but for the default communications device. Everything applies, except you'll be looking for, '-inputcomms' in the command line options section.

Sound Effects

Click the, 'Sound Effects' button to open up the, 'Manage Effect Sets' screen. From this screen, you can manage all of your sound effect sets. Sound effect sets in VoiceAttack are collections of sound effects that can be applied to text-to-speech phrases to hopefully make them a little more interesting.



The, ‘Manage Effect Sets’ screen lists each of your effect sets. You can add, edit and remove effect sets as well as export and import effect sets from here. To export an effect set, simply click the export button (the icon is the arrow that points to the upper-right). Just choose a file name and location and your effect set will be exported as a .vfx. An exported effect set (.vfx) is simple XML data (author flags coming soon). Importing an effect set is just as easy. Just click the import button (the icon is the arrow that points to the bottom-left), select the .vfx that you want to import and you’ll be good to go. Adding or editing an effect set (the, ‘plus’ icon and ‘pencil’ icon) will open up the Effect Set editor screen. This screen allows you to specify which effects comprise your effect set. To add an effect to your set, just click on the button with the, ‘plus’ icon. You can currently choose to add echo, distortion, phaser, flanger, chorus, reverb, auto wah and pitch shift. You can reorder the effects by clicking the up and down arrows. Depending on the selected effects, reordering the effects in your set will sometimes alter the final result. Each effect type has its own group of settings for you to experiment with - just move the sliders to find out what each can do. You can also disable/enable individual effects in your set if you need to by checking (and unchecking) the, ‘Disable’ box. To remove an effect from the set, just click the button with the, ‘X’ icon. To hear what your set will sound like when it’s applied, just click the, ‘Preview’ button. You can change the preview text and volume as well as the text-to-speech voice.

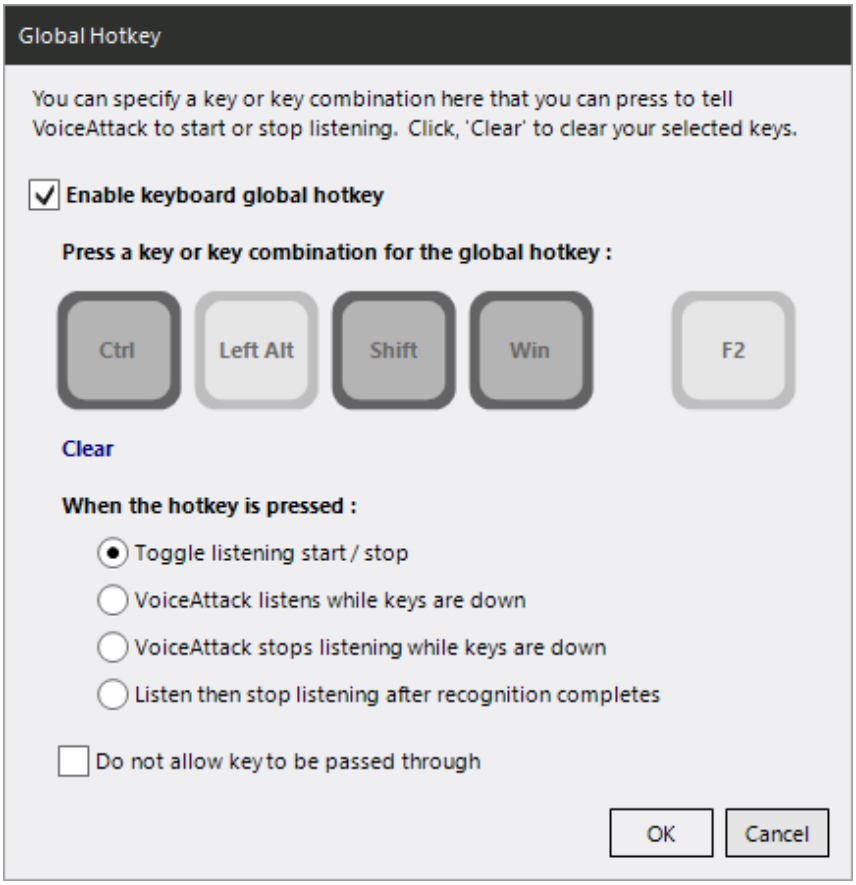
Note that in order for VoiceAttack to apply your effect sets, you’ll need to have,

‘Integrated Components’ selected in Options > Audio > Audio Output Type.

Hotkeys Tab

Recognition Global Hotkey

This is the key/key combination that stops and starts VoiceAttack's, 'listening'. You can make VoiceAttack toggle listening on and off, or, you can hold down this key/key combination to make VoiceAttack listen and more. The default value for this is to toggle listening with Alt + F2 (works just like clicking on the Listening/Not Listening button on the Main screen. To change this value, click the, '…' button, and, you will be presented with the following screen:



From this screen, you can enable/disable this key combination. To choose a key combination, simply hit the keys that you want to use. Modifiers (shift/alt/ctrl/win) will highlight on the left, while the main key will highlight on the right. To clear all of the keys, just click the, 'clear' link.
The listening methods are as follows:

Toggle listening start/stop - Choose this option if you want VoiceAttack to toggle listening on and off with each press of the key/key combination.

VoiceAttack listens when keys are down – VoiceAttack's listening is only enabled when the key/key combination are held down.

VoiceAttack stops listening while keys are down – VoiceAttack's listening is always enabled, except for when the key/key combination is held down.
Listen then stop listening after recognition completes – Also known as, ‘Listen Once’, this option enables VoiceAttack's listening until the next speech event completes (either recognized or unrecognized) and then listening is automatically disabled.

The, 'Do not allow key to be passed through' option prevents the main key (non-modifier) from being passed through to the focused application. For example, if your hotkey is F1 and this option is selected, VoiceAttack will respond to the F1 key press and then prevent any other application from receiving this key press (for this example, if F1 is being handled by VoiceAttack, you will not be able to use the F1 key while other applications are running. If you rely on F1 to bring up, 'Help', then, you'll have to pick another key). **Use care when selecting this option.**

Note: If the key is part of a combination and this option is selected, hitting the main key by itself will still pass through to the application.

Note: This option can be overridden at the profile level. See, 'Profile Options' screen.

Mouse Click Recognition

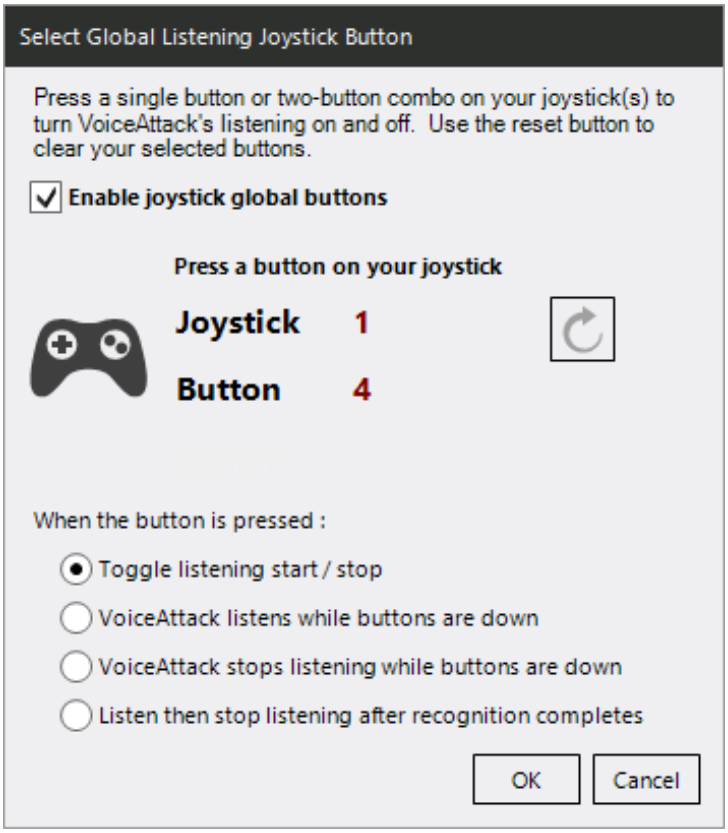
Works like the Global Hotkey, only with your mouse. You can toggle 'Listening/Not Listening', ‘Listen until recognition’, hold the mouse button to listen and hold the mouse button to stop listening. This feature is available on the five standard mouse buttons (left, right, middle, forward & back).
Note: This option can be overridden at the profile level. See, 'Profile Options' screen.

Stop Command Hotkey

This is the hotkey you can select to act as a panic button to halt all running macros. This works the same as pressing the 'Stop Commands' button on the main screen (See, 'Main Screen'). Click the, '…' button to change this option (this works pretty much the same as the, 'Global Hotkey', so, for brevity, the screen shot and description is omitted).
Note: This option can be overridden at the profile level. See, 'Profile Options' screen.

Joystick Button Recognition

This is the joystick button that stops and starts VoiceAttack's, 'listening'. You can make VoiceAttack toggle listening on and off, or, you can hold down this button to make VoiceAttack listen and more. To change this value, click the, '…' button, and, you will be presented with the following screen:



From this screen, you can enable/disable this joystick button. To choose a button, simply press the button on the joystick that you want to use. To clear the button, just click the, 'reset' icon in the top-right.

The listening methods are as follows:

Toggle listening start/stop - Choose this option if you want VoiceAttack to toggle listening on and off when the selected buttons are pressed.

VoiceAttack listens while button is down - VoiceAttack's listening is only enabled when the selected button is held down.

VoiceAttack stops listening while the button is down - VoiceAttack's listening is always enabled, except for when the selected button is held down.

Listen then stop listening after recognition completes – Also known as, ‘Listen Once’, this option enables VoiceAttack's listening until the next speech event completes (either recognized or unrecognized) and then listening is automatically disabled.

Note: This option can be overridden at the profile level. See, 'Profile Options' screen.

Keyboard shortcut double tap threshold (ms)

This option indicates the maximum amount of time (in milliseconds) between key presses to indicate that a double tap shortcut has occurred. For instance, if you have a command set up with, say, F12 as a double tap shortcut, this option is the maximum time allowed between the first time you press F12 and the next time you press F12. If you press F12 within this time limit, VoiceAttack will register the two presses together as a double tap. If you do not press F12 within this time limit, VoiceAttack will consider each press of F12 as a single press. The default value for this option is 300 milliseconds. Decrease this number if your key presses are generally pretty fast. Increase this number if you need more time between key presses.

Mouse shortcut double tap threshold (ms)

Joystick button shortcut double tap threshold (ms)

These work just like the, ‘Keyboard shortcut double tap threshold’ option above, just with mouse and joystick buttons, respectively.

System/Advanced Tab

Bypass Mouse Targeting

This option disables setting the process target of the application located under the mouse on a mouse-down event. When this option is not checked, when a mouse-down event occurs (mouse down, click, double-click) the application located under the mouse is selected as the process target (overriding any set process target) for the duration of the command.

Single TTS Instance

VoiceAttack will allow for essentially any number of text-to-speech (TTS) instances by default. Some TTS packages are particular about the number of instances of their voices that are running and will cause VoiceAttack to hang or crash. Check this option if you are experiencing trouble with a TTS voice package.

Cancel Blocked Commands

Checking this option will prevent commands from executing that are blocked by synchronous commands (commands that do not have the option, 'Allow other commands to be executed while this one is running' checked). Deselecting this option will cause any commands that are triggered during the block to be executed when the blocking command completes. Note that there is no guarantee of the order of the execution of the blocked commands and there is also no guarantee that the blocked commands will themselves be able to block. Hope that makes sense o_O.

Show Third-Party App Warnings

Some third-party applications can cause conflicts with VoiceAttack. Selecting this option will show a warning in the log at the startup of VoiceAttack if any of the predefined applications are running. The list of applications will be updated when you click on the, ‘Check for Updates’ button. This value is on by default, so if you get sick of seeing warning messages when you start up, deselect this option.

Stop Running Commands When Editing

When this option is selected, any commands that are running when the Profile Edit screen is opened will be stopped. This will also prevent any commands from running while the screen is open. Deselecting this option will allow commands to continue while the profile is being edited and will not stop commands from being executed. Use with caution when turning off this option.

Use Nested Tokens

When this option is selected, tokens are processed from the inner-most token to the outer-most token, reprocessing from the inside out on each iteration. This allows for tokens to be nested, or, in other words, tokens can provide input values to other tokens. This may possibly cause issues with older commands. Unchecking this box puts VoiceAttack back to what it used to be and may help if some tokens are giving you problems. Note that this is a global setting, so you will want to fix up your old tokens if you want to be able to use this functionality.

Allow Command Segment Info for Composite Commands

This option will allow you to capture command segment information when using composite (prefix/suffix) commands. Capturing command segment information requires a level of processing that may be noticeable when large numbers of composite commands are created that use dynamic command sections. Uncheck this box if you do not intend to use command segments with composite commands (see the, '{CMDSEGMENT]' token for more information).

Upon import, profiles will have, 'Block potentially harmful profile actions' selected

If this option is selected, each profile that is imported will have its 'Block potentially harmful profile actions' option turned on. This will help keep a variety of items from being run from within a profile prior to inspection. See the, 'Profile Options' screen for more details.

Enable Log Quiet Mode

This option controls, in some part, what information is sent to the VoiceAttack log. When this option is checked, nothing is displayed in the VoiceAttack log unless there is an entry generated by VoiceAttack that is the result of an error (indicated by a red dot). Selecting the, 'Display Warning Log Items' option will allow warnings to also be displayed (indicated by a yellow dot). The, 'Display 'Write To Log' Actions' option will allow any action that writes to the log to be displayed.

Run VoiceAttack as an Administrator

This option will make VoiceAttack attempt to run as an administrator. Note: If you find yourself in a situation where you cannot get into VoiceAttack to uncheck this setting, you'll want to use the, '-clearasadmin' command line parameter (which clears the setting).

Disable Audio Prefetch

This setting will allow you to override any profile that has, 'Play a Sound' actions that have the option of, 'Prefetch' turned on. Note that profiles that are prefetching certain audio files are probably doing so for performance reasons. Only set this value if you are experiencing slow profile load times.

Preserve Prefetched Audio on Profile Change

Normally, switching profiles will cause all prefetched audio to be cleared. Select this option to retain prefetched audio between profile changes. This will boost performance somewhat if you switch profiles frequently. Note that prefetching audio and not clearing it can cause VoiceAttack to run out of memory (and subsequently causing it to crash). Use this option with caution.

Reset Screens Button

Pressing this button will allow you to reset the position and size of all VoiceAttack screens.

Use Built-In SAPI Speech Engines / Use Installed Speech Platform Speech Engines

If these options are enabled and available for you, that means that you have both the built-in SAPI speech engine (the one that is installed with Windows), and at least one Microsoft Speech Platform speech engine installed on your computer. This option lets you toggle between the two speech platforms. Note that if you change this option, VoiceAttack must restart before the speech platform will switch. You can learn more about installing the Microsoft Speech Platform 11 speech engines by clicking the link below the options.

Use Built-In SAPI Text-To-Speech / Use Installed Speech Platform TTS Synthesizers

If these options are enabled and available for you, that means that you have both the built-in SAPI speech engine (the one that is installed with Windows), and at least one Microsoft Speech Platform text-to-speech synthesizer installed. This option lets you toggle between the two platforms. Note that if you change this option, VoiceAttack must restart before the speech platform will switch. You can learn more about installing the Microsoft Speech Platform 11 speech engines by clicking the link below the options.

Enable Sleep Mode

When the Windows speech engine is active, Windows is prevented from going to sleep on its own. When this option is enabled, VoiceAttack will attempt to put the speech engine to sleep (‘Sleep Mode’) if no audio is detected after the number of seconds you specify. When the speech engine is off, Windows will then be able to sleep if it is able. To wake up the speech engine out of, ‘Sleep Mode’ just move the mouse, click a mouse button or press a keyboard key.

Prevent Speech Engine From Changing Microphone Volume

This option attempts to keep the speech engine from changing the microphone volume when the speech engine is started up. The reason that the microphone volume is altered by the speech engine is due to the environment at the time that you had trained the speech engine. That is, the speech engine is attempting to compensate for an environment that was either too quiet (by raising the mic volume) or too loud (by lowering the mic volume). Generally, this is a good thing, and it is recommended that you do not mess with this setting. If the setting is too far out of comfortable range, it is recommended that you train your speech engine properly in an environment that is more speech engine-friendly (you, know… quiet and without kids bouncing around). However, sometimes you need to be able to just go all draconian and bypass whatever

is in place to see if it helps, so, this setting is for that situation. ***Note that this is a Windows-level setting, and any other instances of the Windows speech engine will be affected by this setting. Note also that this feature is not available to all versions of Windows.***

Auto-Adjust CPU Utilization During Command Execution

Changes to Windows 10 (version 2004+) have affected the way applications handle timing in certain ways, including VoiceAttack. If you are using Windows 10 version 2004 and up, VoiceAttack will adjust its timing automatically, which in turn will increase performance while running commands. It is recommended that you leave this setting enabled and only disable if your existing commands are suddenly running too fast. In that case you'll only want to disable this feature until you've made necessary adjustments to your setup.

Export Settings Button

This will allow you to save VoiceAttack's settings to a single file that you can maintain as a backup. Note that registration details and profile information are not stored in this file.

Import Settings Button

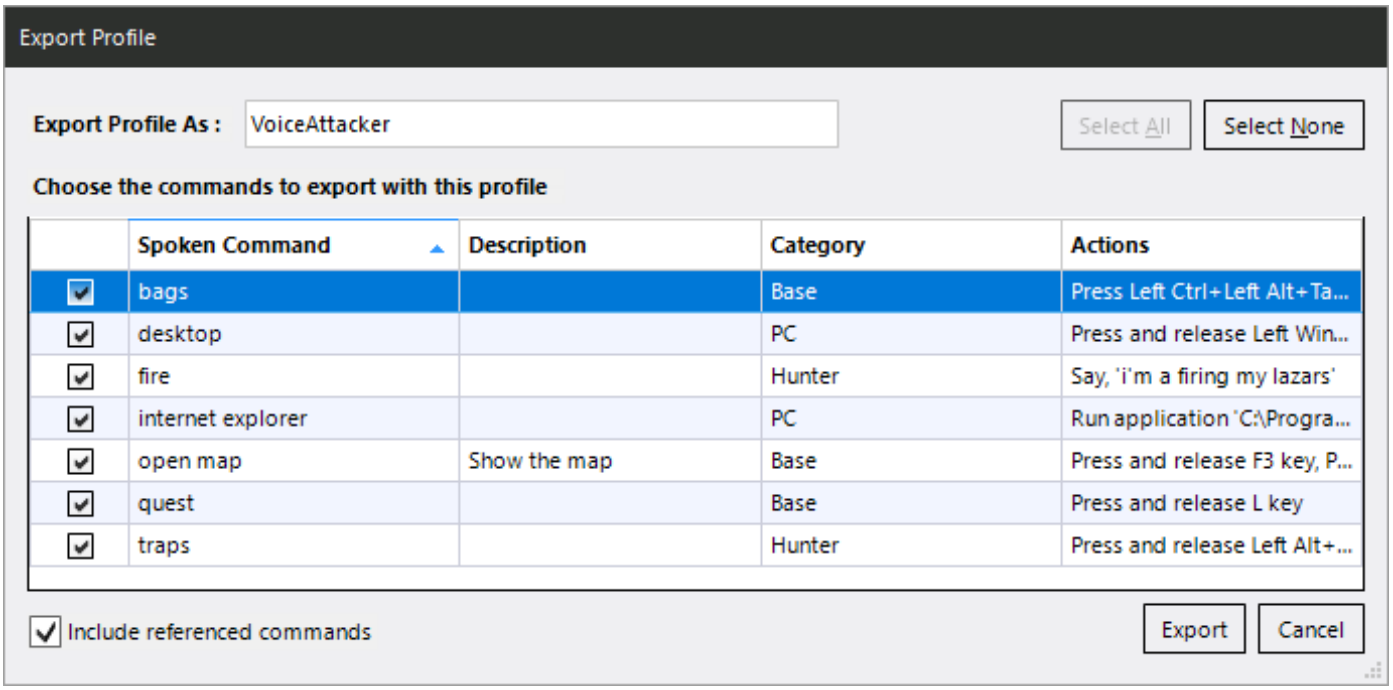
Use this feature to import your previously-saved settings file to restore your settings. Note that this will not affect registration details or profile data and VoiceAttack will close and need to be restarted after your settings are imported.

Browse VoiceAttack's Data Folder Link

Click this link to browse the folder that holds VoiceAttack's central data file and backup folder. See the section labeled, 'Voice Attack's Data Storage' later in this document.

Exporting Profiles

Exporting a VoiceAttack profile is very simple. Just click the 'Export' button on the main screen (See 'VoiceAttack's Main Screen') and you will be presented with the Export Profile screen below:



Next, you can give your exported profile a new name by typing it into the 'Export Profile As' box. You will notice that all of the available commands are checked. If you do not want certain commands exported, simply deselect the boxes next to them. When you are ready to go, click on the 'Export' button, and, you will be presented with a 'Save Profile' dialog box, where you will have the choice of three different exported types. The first type, '**VoiceAttack Profile**' is probably going to be your most-frequently used choice. This produces the most compact exported file size. The second item, and probably second-most frequently used will be, '**Quick Reference List as HTML**'. This will export an HTML representation of your profile that you can view or print with a browser (like Internet Explorer or Chrome) (see the next section labeled, 'Creating Quick-Reference Lists' for more info on that). The third option, '**VoiceAttack Profile Expanded as XML**' will export your profile to an XML-based file that you or anybody can later edit or examine using a text editor (like Notepad). Note that the size of the exported file produced by this option can be relatively large.

The option, 'Include referenced commands' will allow you to include (by checking) or exclude (by unchecking) referenced commands with your export. A referenced command is a command that is either executed by an, 'Execute Another Command' action or is terminated by a 'Stop Another Command' action. Note: There are very few instances where you would want to exclude referenced commands. Note also that commands that are included from other profiles will not be available to export from the current profile. **Known Limitation** - If a command is referenced using a phrase derived from dynamic

command sections (that is, using ‘test 1’ to reference ‘test [1;2]’), that command will not be included when exporting. You will want to make sure you explicitly select dynamic commands to be exported if they are referenced in this way.

To export your profile to share or to make a backup, make sure that you have selected either 'VoiceAttack Profile' or ‘VoiceAttack Compressed Profile’ in the, 'Save as Type' box (see above for a description of each). Pick a good place to save and hit the, 'Save' button. All saved VoiceAttack profile files are given the '.vap' (VoiceAttack Profile) extension. To import this newly saved profile, see the 'Importing Profiles' section later in this document. To import individual commands from this exported profile, see, 'Importing Commands' (also later in this document).

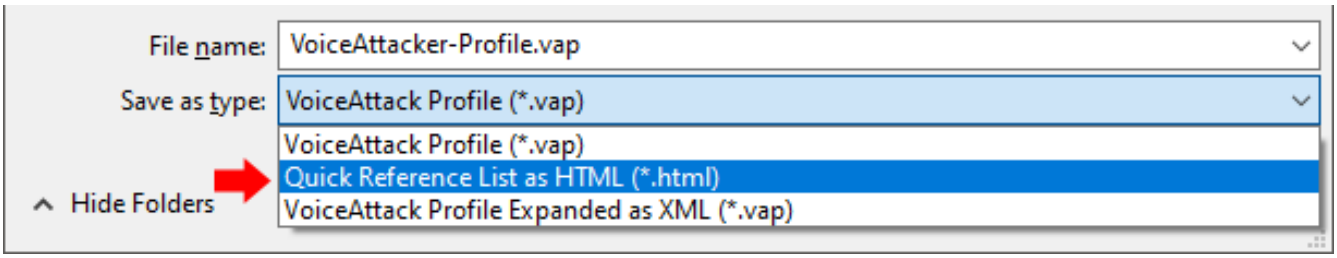
NOTE: The Export Profile functionality is only available for registered versions of VoiceAttack.

Creating Quick-Reference Lists

Creating a Quick-Reference List like the one below uses the same steps as exporting a profile.

VoiceAttacker	VoiceAttack Profile Command Sheet
Spoken Commands	**Actions**
bags	Press and release SHIFT+B keys
desktop	Press and release Windows+D keys
fire	Say, 'i'm a firing my lazars'
focus one	Press and release CTRL+ALT+SHIFT+1 keys
focus three	Press and release CTRL+ALT+SHIFT+3 keys
focus two	Press and release CTRL+ALT+SHIFT+2 keys
map	Press and release M key
quest	Press and release L key

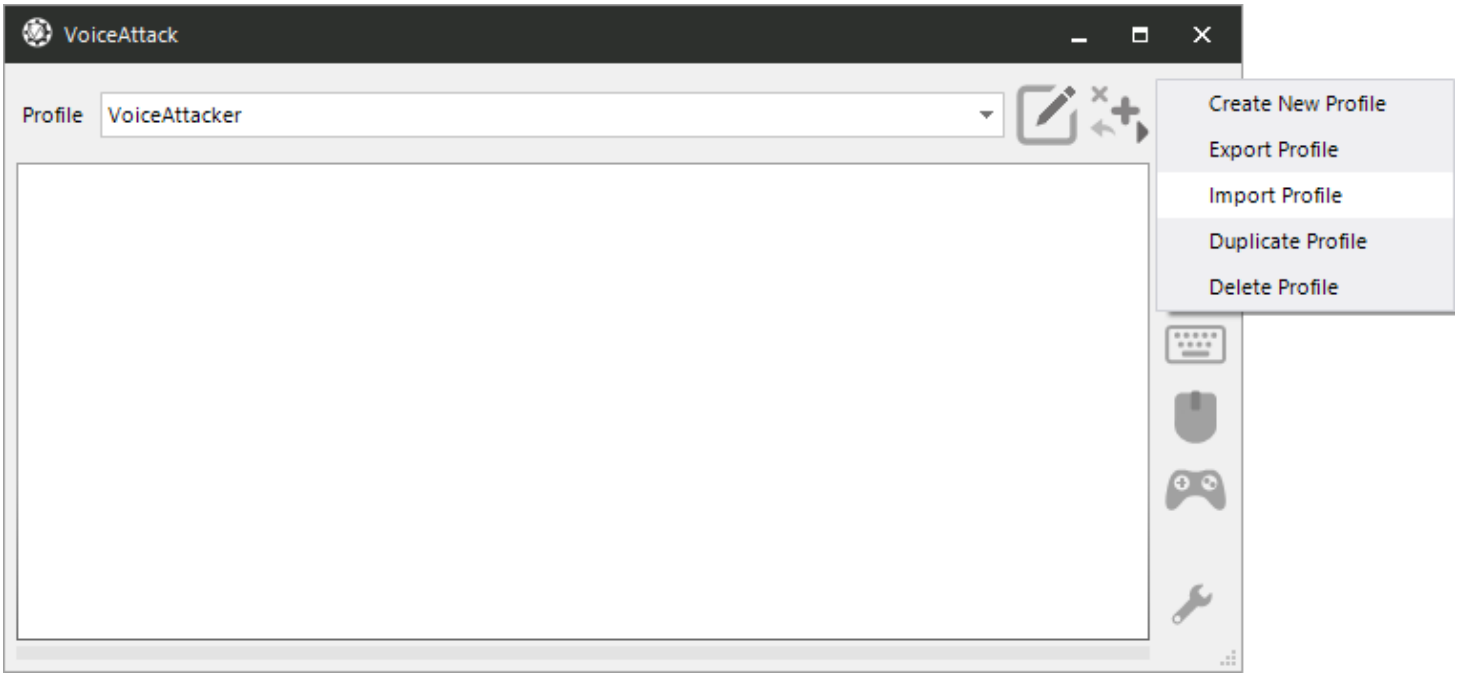
The only difference is that, instead of choosing to save your profile as a VoiceAttack Profile (.vap), you need to save your profile as an HTML file (see image below). The HTML file that is generated is viewable and printable in most compatible browsers.



NOTE: The Export Profile functionality (which includes creating quick reference lists) is only available to registered versions of VoiceAttack.

Importing Profiles and Profile Packages

Importing profiles into VoiceAttack is even easier than exporting ('See Exporting Profiles'). Just go to the main screen, click on the multi-function icon and select, ‘Import Profile’ from the menu (or, press ALT + I):



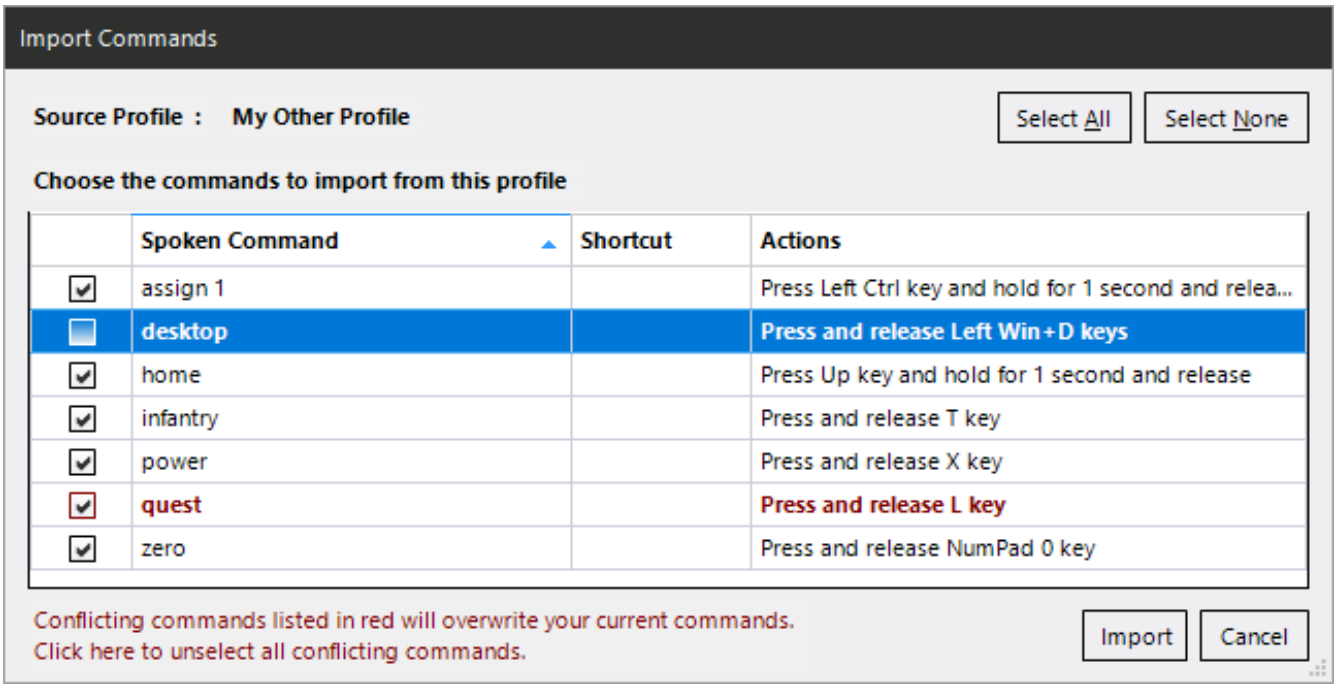
When the 'Select a VoiceAttack Profile File To Import' dialog appears, simply browse and select your previously saved VoiceAttack profile file. A VoiceAttack profile file will have an extension of, ‘.**vap**’.

If the profile being imported has the same name as a profile that you already have, VoiceAttack will rename the new profile for you.

Another type of file that can be imported is the, ‘VoiceAttack Profile Package’ file type. These files have an extension of, ‘.**vax**’. A VoiceAttack Profile Package can contain multiple profiles, sound files, and plugin/application files. When a package file is imported, all of its contents are imported at one time (instead of having to manage multiple files). Each profile it contains is imported (and renamed if duplicated as indicated above). Then, each sound and app/plugin will be copied to their respective locations in the folders that are specified on the Options screen (see, ‘Sounds Folder’ and, ‘Apps Folder’ portions of the, ‘Options Screen’ section earlier in this document). Note that importing a package file **WILL OVERWRITE** any existing sounds or app/plugin files that have the same path and file name, so make sure you know the source of your package and read their provided documentation on where the imported files will be placed. Since plugins/apps can be overwritten, **the plugin feature must be disabled on the Options screen first**, otherwise the import will not run (since files may be in use). For more information on VoiceAttack Profile Package files (as well as how to create them), see the section, ‘VoiceAttack Profile Package Reference’ later in this document. **NOTE**: The 'Import Profile' functionality is only available to registered versions of VoiceAttack.

Importing Individual Commands

To import commands into a profile, first click on the 'Import Commands' button located at the bottom-left corner of the 'Add Profile' or 'Edit Profile' screens (See 'Profiles' section). You will then be presented with an open file dialog titled, 'Select a VoiceAttack Profile Containing Commands to Import'. Browse and select a previously saved VoiceAttack profile (VoiceAttack profile files have a, '.vap' extension). The, 'Import Commands' screen will then appear as below:



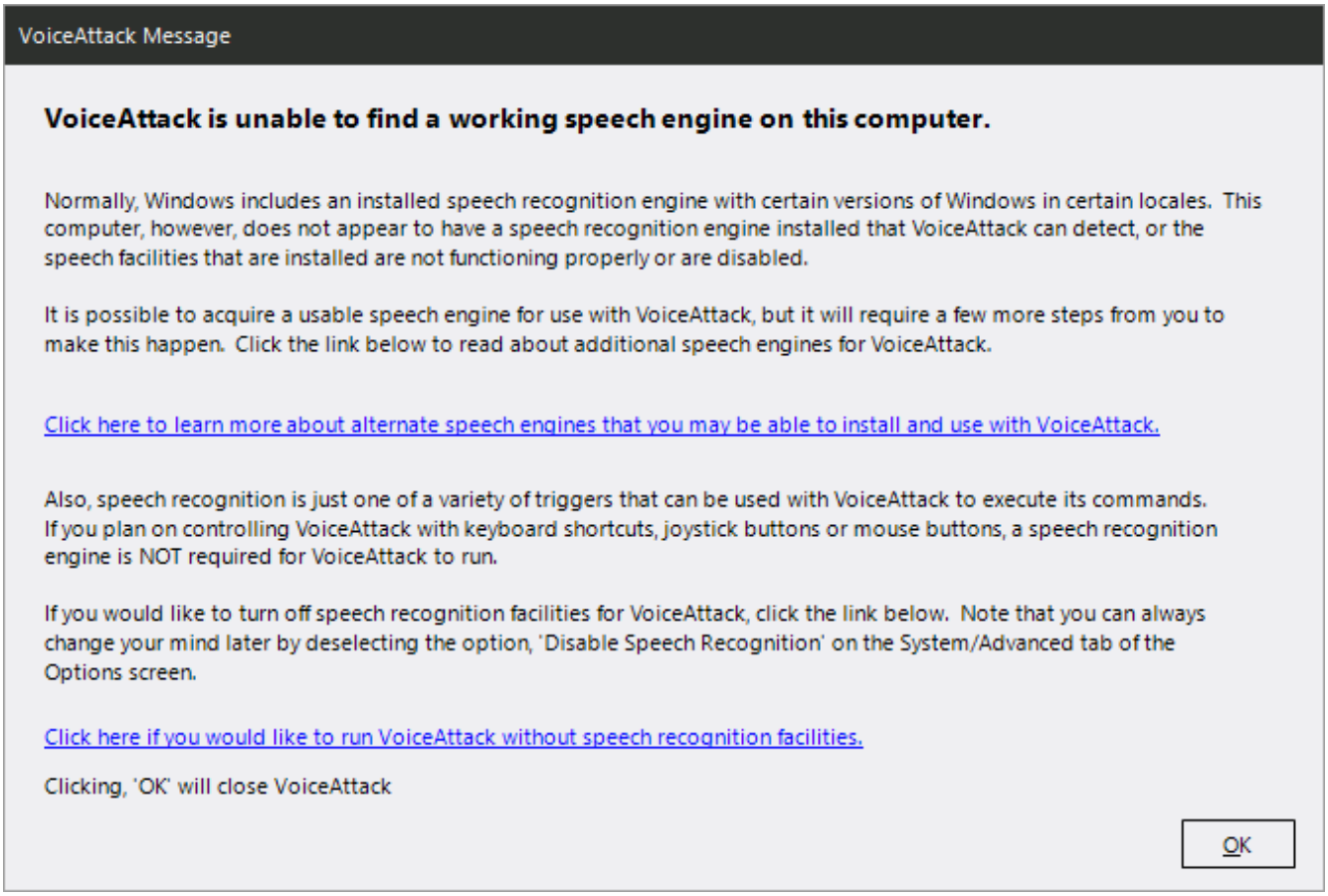
Only the commands that are checked will be imported into your profile. Commands that are in **red** are conflicting commands that already exist in your profile. **Importing the conflicting commands will result in your profile's commands being overwritten**. To clear all conflicting commands at once, simply click the gigantic label in the bottom-left corner :) When you are ready to import the selected commands, click on the, 'Import' button at the bottom-right of the screen. Remember, the commands that you import are not committed until you hit, the 'Done' button on the profile screen.

NOTE: In the unregistered version of VoiceAttack, the single profile given is limited to 20 commands. If the profile ends up with more than 20 commands, only the first 20 will be shown (you may lose commands you have entered, since they may drop off).

No Speech Engine / Alternate Speech Engines in VoiceAttack

To be able to issue voice commands in VoiceAttack, a working speech engine is necessary. For most users with a Windows installation in English, German, French, Spanish, Chinese or Japanese locales, a speech engine in the appropriate language will be readily available. For others, Windows may not come with a speech engine installed, or their installation of Windows may default to an English speech engine.

If no speech engine is detected when VoiceAttack is launched (no speech engines installed on your computer or errors are encountered when asking Windows for its speech engines), VoiceAttack will present you with a message that indicates this. You will then have the option to run VoiceAttack without a speech engine, or, you can follow a link that will guide you on how to install alternate Microsoft (Microsoft Speech Platform) speech engines:



If you would like to run VoiceAttack without a speech engine, simply click the link. VoiceAttack will start up as normal, but voice commands will be completely disabled (even if you have a working speech engine). In this mode, you can still execute VoiceAttack's commands via keyboard shortcuts, mouse button clicks or joystick button presses. This mode is also handy in situations where you just can't use voice commands, such as at work (which I use a LOT, as my coworkers won't miss a chance to tease… "KILL EXCEL!" lol). The next time VoiceAttack launches, no check will be made for a speech engine, as the option is saved once you click the link. To turn speech recognition facilities back on again, simply deselect

the, 'Disable Speech Recognition' check box on the, 'Options' screen (System/Advanced tab).

If you are without a speech engine (or want to try a new speech engine, either out of sheer need or even curiosity), the latest versions of VoiceAttack will allow the use of the Microsoft Speech Platform 11 speech engines. The link on the message will take you to the VoiceAttack site that will guide you on how to download the necessary files to make this happen. Simply follow the instructions provided on the site (which basically shows you how to download the runtime and one or more speech engines/text-to-speech synthesizers), and once VoiceAttack is restarted, the newly-installed speech engines and/or text-to-speech synthesizers will be available for use. If you have both the original (built-in) speech engine(s) and would like to toggle between the two speech platforms, there are options on the System/Advanced tab of the, 'Options' screen. Just choose the platform you wish to use (speech engines or text-to-speech synthesizers (or both)) and click, 'OK'. The next time VoiceAttack launches, you will be able to select from the speech engines or text-to-speech synthesizers from the selected platform (note that when you do change, the selected speech engine reverts to, 'System Default'). This is all new for VoiceAttack, so, once again, come and visit the VoiceAttack User Forums for more discussions on this (and, if there are none, start one!).

Using the Condition Builder

The Condition Builder screen allows you to create compound conditions. Compound conditions contain multiple, single-test conditions that are strung together by, 'AND' or 'OR'.

For instance, a single-test condition (like VoiceAttack has used for some time) looks like this:

```
If myVariable Equals 1 Then
    Do Something
End If
```

This works great, but if you want to check if myVariable is 1 **OR** 2 **OR** 3 in VoiceAttack, you would have to do something like this:

```
If myVariable Equals 1 Then
    Do This
    Do That
    Do Other
Else If myVariable Equals 2 Then
    Do This
    Do That
    Do Other
Else If myVariable Equals 3 Then
    Do This
    Do That
    Do Other
End If
```

Note the duplicated script... there are ways to make this shorter, but that’s another subject :)

Additionally, if you wanted to check if myVariable was 15 **OR** myVariable was between 1 and 10, but did not equal 6, you would need to do something like this:

```
If myVariable Equals 15 Then
    Do This
    Do That
    Do Other
Else
    If myVariable is Greater than or Equals 1 Then
        If myVariable is Less than or Equals 10 Then
            If myVariable Does Not Equal 6 Then
                Do This
                Do That
                Do Other
            End If
        End If
    End If
End If
```

As you can see, things can get a little wordy, and these are actually just small examples without a lot going on. It would be better to have scripts that read like this:

```
If myVariable Equals 1 OR myVariable Equals 2 OR myVariable Equals 3 Then
    Do This
    Do That
    Do Other
End If
```

And

```
If (myVariable Equals 15) OR (myVariable >=1 AND myVariable <= 10 AND myVariable Does Not
Equal 6) Then
    Do This
    Do That
    Do Other
End If
```

To build these kinds of statements in a VoiceAttack action/graphical fashion requires the use of the Condition Builder. The Condition Builder looks scary, but once you get an idea of what it's doing, you'll get the hang of it quickly. To understand what is going on, let's start with some terminology. My apologies in advance to any programmer types out there ;)

First is the term, '**condition**'. In programming, this is often referred to as an, 'If' statement. In VoiceAttack, you'll see conditions are used for, 'If' statements, 'Else If' statements, and 'While' loops:

```
If myVariable Equals 5 Then
    Do Something
Else If myVariable Equals 6 Then
    Do Something Else
End If

Start Loop While:  myVariable Is Less Than 10
    Do Stuff
    myVariable = myVariable - 1
End Loop
```

A, '**compound condition**' is an expression that contains more than one condition. The conditions are strung together by, '**AND**' or, '**OR**' and consist of logical '**sets**':

```
If (myVariable Equals 15) OR (myVariable >=1 AND myVariable <= 10 AND myVariable Does Not
Equal 6) Then
    Do Something
End If
```

The first logical set above is '**myVariable Equals 15**'. If the myVariable integer variable is 15, the entire expression is evaluated as true and the control moves to the next line.

If myVariable' value is 6, the first set does not evaluate to true, so the second logical set '(myVariable >=1 AND myVariable <= 10 AND myVariable Does Not Equal 6)' is evaluated. The second set also does not evaluate to true, which means the compound condition evaluates to false and control moves past the end of the compound condition (past the, 'End If').

Again, in VoiceAttack, you can build these, 'sets' using the Condition Builder. The expressions resulting from the Condition Builder can be one or more logical sets. Each set is evaluated within the expression. **If all the conditions in a set evaluate to true, the set is true**. **If any of the conditions in a set evaluate to false, the whole set evaluates to false**. **If ANY set evaluates to true, the entire expression evaluates to true**. Another way to state this is everything WITHIN a set is, 'AND-ed' and all sets are, 'OR-ed':

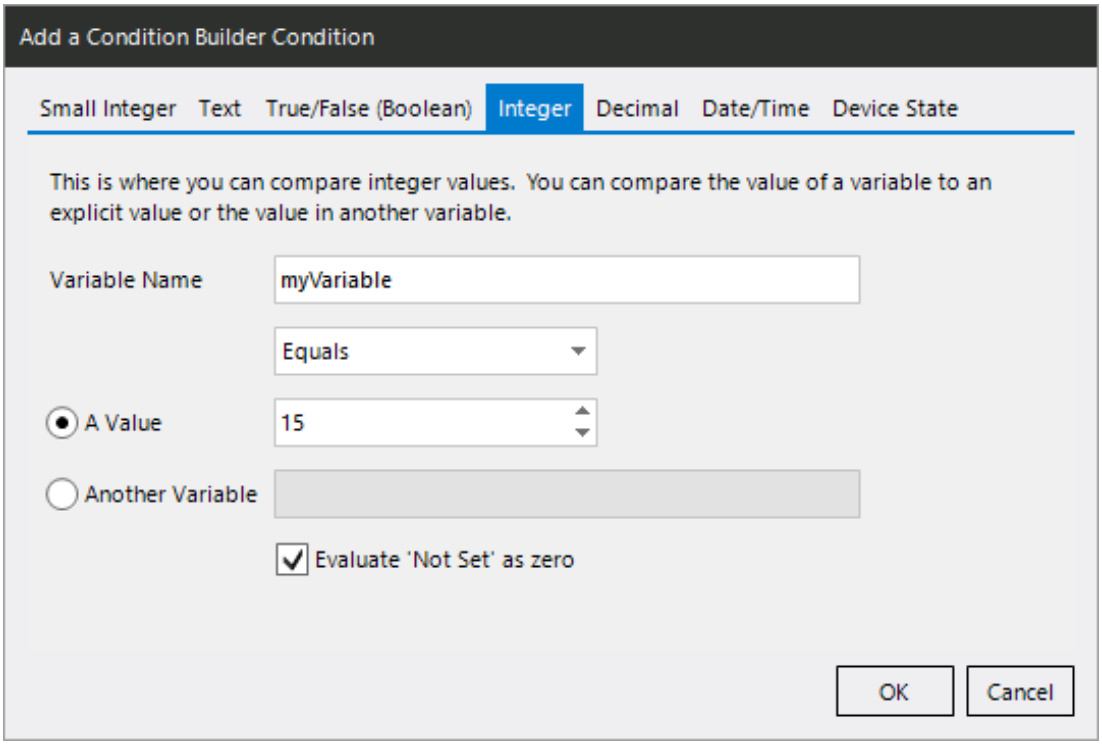
<u>Set 1</u>
myVariable Equals 15

OR ←

<u>Set 2</u>
myVariable >=1
AND ←
myVariable <= 10
AND ←
myVariable Does Not Equal 6

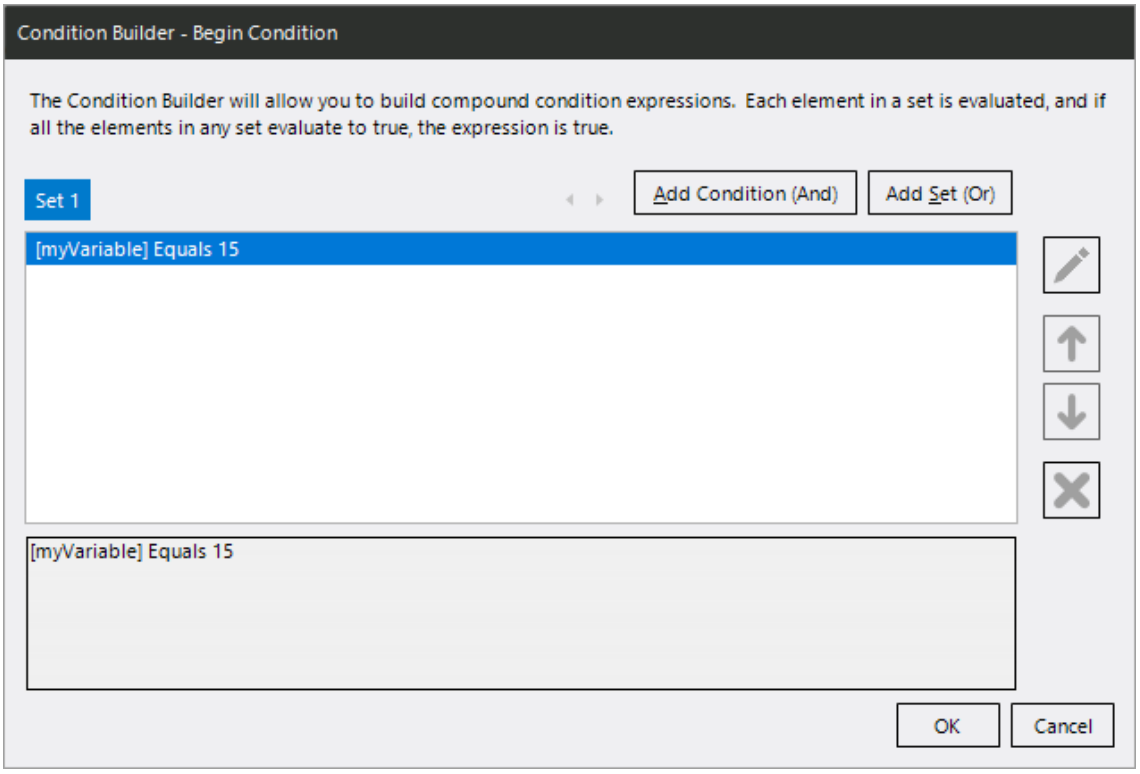
The best way to explain how this is done in VoiceAttack is to just jump right in:

First, from the Command screen, select, `Other > Advanced > Begin a Conditional (If Statement) Block > Compound Condition Builder`. You will be presented with a familiar screen to create the first condition. Click on the, 'Integer' tab and then enter the values as you would to create the condition, 'myVariable Equals 15':



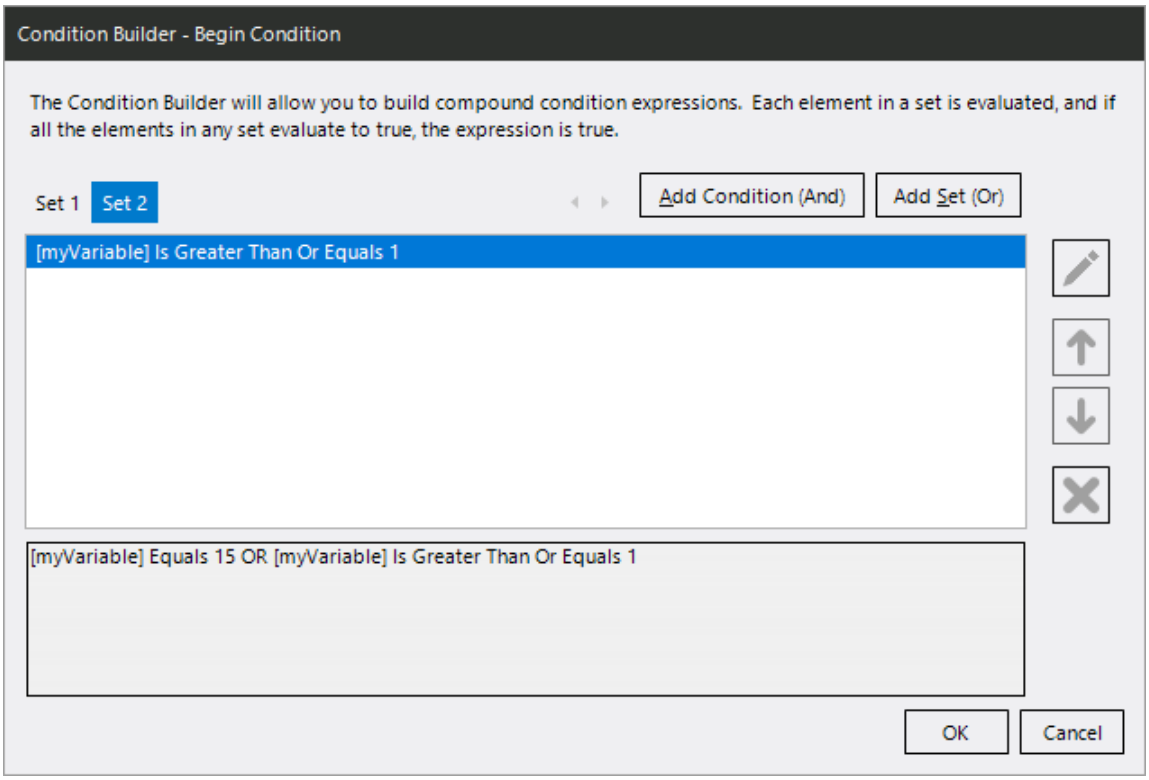
Click OK, and you'll notice that the first set (Set 1) is created for you with the condition,

'myVariable Equals 15' as the first item. Note the narrative at the bottom of the Condition Builder screen indicates the current state of the entire expression:



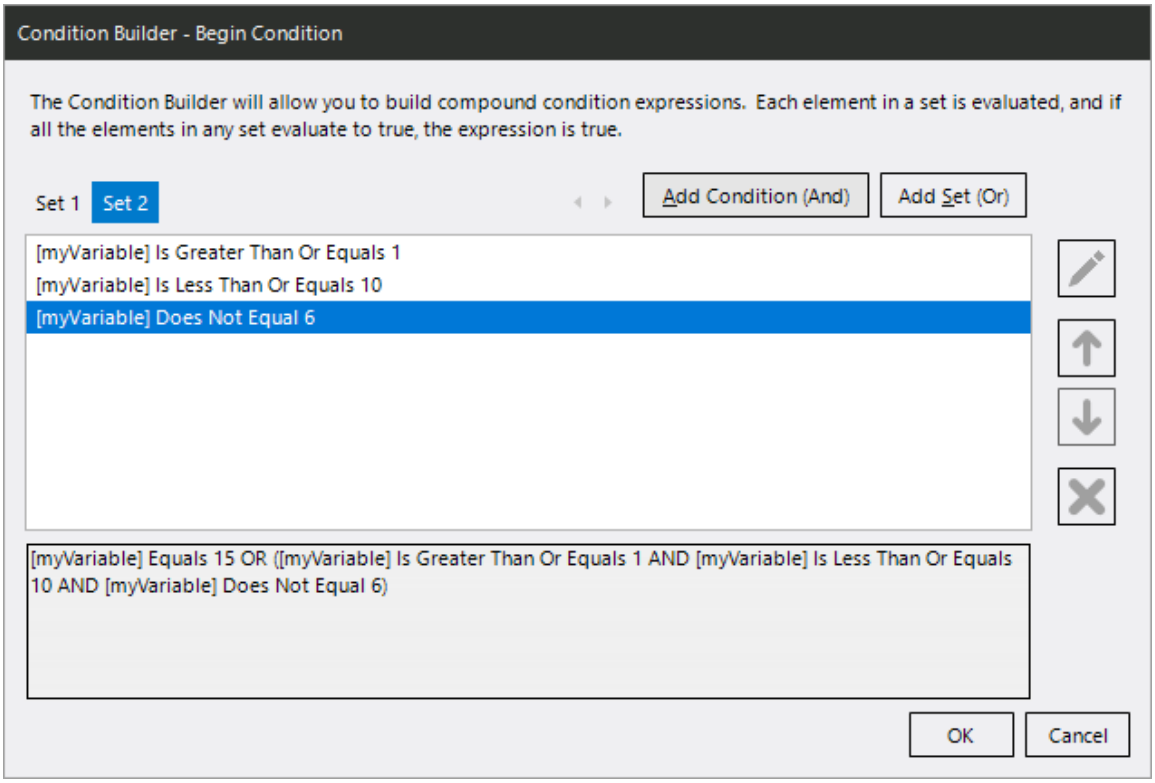
That's all there is for the first set... we're just checking first to see if myVariable is 15.

Next, we'll want to create the second, 'Set'. Click on the, **'Add Set (Or)'** button. Again, you'll get an, 'Add a condition' screen popped up to add the first condition of the new set. Click on the, 'Integer' tab and, again, enter the values as you would to create the condition, 'myVariable is greater than or equal to 1' and click, 'OK'. Note that you are now in, 'Set 2', with 'myVariable is Greater than or Equals 1' as the first condition on the Condition Builder screen. If you look at the narrative at the bottom of the screen, you should see, '`myVariable Equals 15` **`OR`** `myVariable is Greater Than or Equals 1`'. Good so far:



We still have two more conditions to add to this set, so, this time click on the, '**Add Condition (And)**' button. You'll be presented again with the, 'Add a condition' screen. Click on the integer tab and enter the values to create the condition, 'myVariable is less than or equal to 10'. Click, 'OK'. A second condition is now added to, 'Set 2': 'myVariable is Less Than or Equals 10', with the narrative now showing, '`myVariable Equals 15 OR (myVariable is Greater Than or Equals 1 AND myVariable is Less Than or Equals 10)`'. One more to go...

Click again on the, '**Add Condition (And)**' button. Again, enter the values to create the condition, 'myVariable Does Not Equal 6' and click, 'OK'. Now, you have three conditions in Set 2, with the narrative now reading, '`myVariable Equals 15 OR (myVariable is Greater Than or Equals 1 AND myVariable is Less Than or Equals 10 AND myVariable Does Not Equal 6)`':



Click, 'OK' on the Condition Builder screen and you'll see your new conditional (if) statement presented in the Command Screen.

Congratulations! You just built your first compound condition. Hope that didn't scare you off :)

Note that you don't have to build compound conditions with all the same data type (in this example we are using all integers). You can have compound conditions based any number of mixed data types.

Note also that you can convert a single-condition conditional statement (you know… the ones you've been using all this time) to compound conditions. Just right-click on the action you want to convert and select the, 'Edit with Condition Builder' option.

Command Line Options

VoiceAttack supports the following command line options (in basically the order in which they were added in case you're wondering).

Note: If VoiceAttack is already started, most command line parameters affect the already-running instance of VoiceAttack. This way you can create desktop shortcuts that affect VoiceAttack. Command line options are case-sensitive.

-listeningoff Turns VoiceAttack's listening off.
-listeningon Turns VoiceAttack's listening on.

-shortcutson Turns VoiceAttack's hotkey shortcuts on.
-shortcutsoff Turns VoiceAttack's hotkey shortcuts off.

-mouseon Turns VoiceAttack's mouse shortcuts on.
-mouseoff Turns VoiceAttack's mouse shortcuts off.

-joystickson Turns VoiceAttack's joystick button shortcuts on.
-joysticksoff Turns VoiceAttack's joystick button shortcuts off.

-minimize Starts VoiceAttack minimized.

-restore Restores VoiceAttack's main screen to a normal window state if minimized, maximized or minimized to the system tray.

-nofocus Prevents VoiceAttack from popping up an already-running instance (if this is not specified, VoiceAttack pops up as the topmost window).

-profile "*My Profile Name"* Changes VoiceAttack's active profile. Note that double quotes are only necessary if your profile name has spaces in it.

-command "*My Command Name"* Executes the command specified by name in the running profile. Note that double quotes are only necessary if your command name has spaces in it.

-stopcommands This will work just like clicking the, 'stop commands' button.

-bypassPendingSpeech If VoiceAttack's 'not listening' mode is invoked, and you are in the middle of issuing a command, VoiceAttack will allow you to finish your phrase and not cut off what you were saying. For some, this behavior is not what is expected. What is expected is that setting VoiceAttack into, 'not listening' mode should cut off immediately, excluding anything that is currently being spoken. To allow VoiceAttack to cut you off immediately, use -bypassPendingSpeech. Note this does require VoiceAttack to be restarted (does not affect an already-running instance).

-ignorechildwindows This is to help with toolbox windows that are always on top. By default, VoiceAttack will seek out popup windows. This attempts to suppress that check. This is experimental right now and may become a feature later with a finer level of control if

the need is there. For now, it is available as a global setting.

-verifyaudio This will make VoiceAttack check all ‘play a sound’ and ‘play a random’ sound files to see if they exist. This just runs at startup and does not affect the already-running instance.

-datadir “*Directory Path”* Allows you to indicate the **directory** of VoiceAttack’s data file (VoiceAttack.dat) and its associated backups. If the VoiceAttack.dat file does not exist in the indicated directory when a profile has been updated, a new VoiceAttack.dat will be created. Note that double quotes are only necessary if your path has spaces in it. This does not affect the running instance.

Example: -datadir “C:\Users\MrAwesome\Desktop\MyData” will look for the VoiceAttack.dat file in the “C:\Users\MrAwesome\Desktop\MyData” directory.

-alwaysontopon This will set the running instance of VoiceAttack to be the top-most application.

-alwaysontopoff This will turn off, ‘Always on Top’ if it is on, returning VoiceAttack to not be the top-most application.

-showloadtime This will show the amount of time it takes to load a profile in the log. The time taken by the speech engine is shown as well. Both times are in milliseconds. This is useful for the folks making enormous profiles to give some sort of indication of what is going on.

-nospeech This disables VoiceAttack speech recognition initialization. This is experimental and functionally incomplete and is used primarily for testing (that is, it is not currently supported). However, this may be a last resort for those that just cannot get their speech engine to work and need to get into the software for whatever reason.

-input *“Device Name”* This will set the Windows default multimedia recording device (table mics and headset mics for example) and set the speech engine to this device. The device name parameter value (between the **required double quotes**) must be spelled exactly as it is shown in the list on the VoiceAttack Options screen (see the Options screen for more information on where to find this). Sometimes with certain events (like a driver update) the name may change. This change may even be very slight (and frustrating). In order to ease that pain a bit, the device name parameter will accept wildcards. For example, this shows how to change to a device exactly as indicated:
C:\Program Files(x86)\VoiceAttack\VoiceAttack.exe -input "Speakers (4- Sennheiser 3D G4ME1)"

This example shows how to select the first device that has a description that **contains**, 'G4ME1':
C:\Program Files(x86)\VoiceAttack\VoiceAttack.exe -input "*G4ME1*"

Search this help document for, ‘wildcards’ for more examples of how they are used.
Note: Double quotes are required for this command line parameter. Also, this parameter

will work against new or existing instances of VoiceAttack.

WARNING: This is not a VoiceAttack setting, rather a Windows device setting and can (and probably will) cause other applications that depend on the changed devices to appear to malfunction. It's not THAT big of a deal, but it will definitely throw you off when your Skype or TeamSpeak is not working how you left them.

-output *"Device Name"* This works exactly like the -input parameter above except it controls the default multimedia output device (like speakers and headphones). Same notes and warning apply as well ;)

-inputcomms *"Device Name"* This works exactly like -input (above), except this option will affect Windows' default communications recording device.

-outputcomms *"Device Name"* Again, this works exactly like the parameters above except it controls the default output communications device (like speakers and headphones).

-inputx *"Device Name"*, **-inputcommsx** *"Device Name",* **-outputx** *"Device Name",* and **-outputcommsx** *"Device Name"* These work exactly like -input, -inputcomms, -output, and -outputcomms except that once the devices are changed, VoiceAttack closes immediately. This means you can make a desktop shortcut or batch file that can change your headphones to your speakers or your tabletop mic to your headset mic (or both at the same time) without fully launching VoiceAttack.

-nokeyboard This keeps all keyboard hooks from turning on when VoiceAttack launches. That means that global hotkeys (such as listening, stop commands, mouse capture) and command shortcuts will be disabled. Note this affects only the instance launched with this parameter and not an instance of VoiceAttack already running.

-nomouse This keeps all mouse hooks from turning on when VoiceAttack launches. That means that global mouse clicks (such as listening) and command shortcuts will be disabled. Note this affects only the instance launched with this parameter and not an instance of VoiceAttack already running.

-uritimeout *Timeout* This indicates the timeout to use when setting text variables via the URI option. This is a stopgap feature until proper user interface is created and may not be in future versions of VoiceAttack. The timeout parameter is expressed in milliseconds: -uritimeout 5000 sets the timeout to 5 seconds. The default value for the timeout that VoiceAttack uses is 30 seconds. Note this affects only the instance launched with this parameter and not an instance of VoiceAttack already running.

-reverseprofilepriority This option will reverse the priority of, 'included' profile commands. By default, when you include commands from other profiles (either from the profile options screen and/or by selecting, 'Global Profiles' from the Options screen), the commands are prioritized first by the active profile, then by the included profiles from the profile options screen, then by the global profiles. With this option, the priority is first the global profiles, then the included profiles from the profile options screen, then the active profile. This is an advanced feature that not many will use, but it is here for those that need it. This only affects the instance launched with VoiceAttack and not an instance that is already running.

-running This option will cause VoiceAttack to write 0 to stdout if no other instance of VoiceAttack is currently running, and 1 if an instance of VoiceAttack is running and then immediately close. Obviously, this does not affect the running instance ;)

-darkon -darkoff These options will turn on and off the dark mode setting for the main screen (‘Cover of Darkness’).

-showcommandnames This option will make the VA log entries display the current command name when a command is executing (for debug purposes).

-asadmin Including this command line parameter will make VoiceAttack attempt to run itself as an administrator for the current session. This does not affect an instance that is already running.

-clearasadmin This option will clear the, ‘Run as an administrator’ setting from the Options screen (this is in case you can’t get into VoiceAttack to clear the setting for some weird reason). This does not affect the instance that is already running.

-resetwarnings This resets various warning messages in VoiceAttack to be visible. This does not affect the instance that is already running.

-priority This adjusts the Windows process priority of VoiceAttack. The values can be, 'low', 'belownormal', 'normal', 'abovenormal', 'high', and 'realtime' (all without quotes). Note that running with a value of, 'realtime' requires VA to be run as an administrator. This only works against a newly-created instance of VA and does not work against an instance that is currently running. Example: **-prority high** runs VA with a high-priority process.

-prefetchstats This will show prefetched audio information, such as the number of audio items prefetched during the current profile load, the total size of the prefetched items during the load and the total time taken to prefetch the items. If the option, ‘Preserve Prefetched Audio on Profile Change’ is selected, the total count and size of all prefetched items will be displayed, as well as the count and size of all preserved items that belong to the current profile.

-noexport This option will allow you to restrict the end user from exporting your profile as binary or XML. That is, if your profile is exported as compressed binary and this command line option was included when VoiceAttack was launched, the exported binary .vap will be set to not allow export after it is imported. That’s a lot to take in for sure, and if you’re not distributing your work, you’ll probably never need to use this.

-punload This indicates the timeout for plugins to unload in milliseconds. When plugins are unloaded on exit, VoiceAttack will wait a specified amount of time before giving up and shutting down. If the timeout is exceeded, an entry will be written to VoiceAttackFault.txt in the installation directory (or data directory, depending on permissions). The default timeout is 5000 (5 seconds). Example setting the plugin unload timeout to one second: C:\Program Files(x86)\VoiceAttack\VoiceAttack.exe -punload 1000

-snapdistance This indicates the distance (in pixels) from the edge of the screen that a

VoiceAttack form must be in order to snap to the edge. The default value is 20 with a maximum value of 500. Set this value to 0 to turn off snapping to edges. Note that this not affect the instance that is already running. Example of how to set snap distance to zero (turning off edge snapping):
C:\Program Files(x86)\VoiceAttack\VoiceAttack.exe -snapdistance 0

-relocate This option will relocate VoiceAttack's open forms to your primary display (as indicated by Windows). This is handy if you've launched VoiceAttack on a display that's either off or disconnected and Windows still thinks that the form should be displayed there ;) Note that this feature can also be invoked from the task bar icon as well as the system tray icon (both labeled as, 'Relocate to Primary Display').

-loadoptions This option will launch VoiceAttack with the, 'Load Options' screen (the same as hold down CTRL + Shift at startup).

-bypassimpropershutdowncheck This tells VoiceAttack to not display the, 'Improper Shutdown' warning message on startup. This should be used only when debugging a plugin and VoiceAttack is shut down improperly as part of an expected process.

Passed Values (Advanced)

Passing values with commands can be done either against the already-running or loading instances of VoiceAttack. There are limitations for both, however. If you are passing values with a command against the already-running instance of VoiceAttack, command-scoped and command-shared values will not be recognized (as there is no calling command to provide this info). If you are passing values with a command against the loading instance of VoiceAttack, only literal values will be recognized, as variables will not yet be available to the command.

Note that if you do not pass a value in to the -Command parameter, these values will be ignored.

The syntax of each parameter below is the same as their 'Execute Another Command' or 'Enqueue Command' counterpart (See, 'Passed Values (Advanced)' as part of the 'Execute Another Command' action earlier in this document):
-PassedText (used for passing text value)
-PassedInteger (used for passing integer values)
-PassedDecimal (used for passing decimal values)
-PassedBoolean (used for passing Boolean values)
-PassedDate (used for passing Date values)

The value part of the parameter should be expressed in **DOUBLE QUOTES**:
-PassedInt "5;myIntVariable"

One thing to note, however, is that in cases where you are required to use them (such as with literal text values), **you will need to escape double quotes**.
To escape double quotes, you will need to put a backslash before them **\"**. For instance, if you want to include the literal value of 'Hello' for the -PassedText parameter, you'll need to

express the parameter like this:

-PassedText "\"Hello\"" <--- Note the backslash + double quotes in there.

or

-PassedText "\”Hello\”";myTextVariable" <--- note the difference between the literal value and the ‘myTextVariable’ variable.

Happy value passing!

Text (and Text-To-Speech) Tokens

A VoiceAttack Token is a tag that can be used as a placeholder to represent a value in places that use text (think of Tokens as mini functions). Tokens are surrounded by curly brackets { } and will always contain a title that is **case-sensitive** : {DATE} or {CMD} or {CLIP}. Tokens started their life in VoiceAttack for use in Text-To-Speech (TTS). Over time, the uses for tokens grew and now they can be used in all sorts of places that require text, such as file paths for launching apps and use in conditional statements. Some tokens you may use a lot, others you may never use at all. Some are simple (like {DATE}) and some are way out there (like {EXP}). An example of how to use the first token ({TIME}) is below. It was used in TTS to help indicate the local time to the VoiceAttack user.

“The current time is now {TIME}”.

This value was put in the, 'Say Something with Text-To-Speech' action. When the action is executed, VoiceAttack replaces the {TIME} tag with the local time (24-hour clock):

“The current time is thirteen hundred hours”.

A variation of the {TIME} token is {time} (notice the lower case). This token will be rendered as a 12-hour clock, so, “The current time is now {time}” would be rendered as “The current time is now one o'clock” when the action is executed.

Some tokens are a little bit more complicated (or flexible, depending on how you look at it) and contain parameters. An example of this would be the {RANDOM:*lowValue*:*highValue*} token. The purpose of this token is to generate a random number within a range.
Notice that this token has TWO parameters: lowValue and highValue to represent the lower and upper boundaries of the range (I believe its first use was to roll virtual, 'dice'). Notice also that the parameters are separated from the title (and other parameters) by colons. As an example, let's say we want to generate a random number between 10 and 20. The token would look like this: {RANDOM:10:20}.

Using Text-To-Speech, your action could use the following statement:

“I am generating a random number and it is {RANDOM:10:20}”.

When the action is executed, VoiceAttack will generate the random number and replace the tag, so the result would be something like, “I am generating a random number and it is 16”.

There are some tokens that take a variable name as a parameter, such as {TXT:variableName}. When VoiceAttack encounters a token that accepts variable names as parameters, it replaces the tag with the value in that variable. In the case of {TXT:variableName}, VoiceAttack will get a text variable's value. {INT:variableName} will get an integer variable's value. {BOOL:variableName} will get a boolean variable's value (and so on). Let's say you set up a text variable called, 'myText' to have a value of, 'Dave'. The token you would use to get the value from 'myText' would be {TXT:myText}. You could use the token in a Text-To-Speech statement such as this:

“I'm sorry, {TXT:myText}, I'm afraid I can't do that.” When the action is executed, the rendered statement would be, “I'm sorry, Dave, I'm afraid I can't do that.” Then, if you set 'myText' to have a value of, 'Sally' and execute the statement again, the rendered statement would be, “I'm sorry, Sally, I'm afraid I can't do that.”.

With version 1.6 of VoiceAttack and later, tokens can be, ‘nested’. That means you can have tokens that accept the value of other tokens. Tokens are processed from the inner-most token to the outer-most token, reprocessing with each iteration. Also, anything contained within curly brackets { } that do not resolve to a token will be rendered as whatever is contained within the brackets. To indicate a literal curly bracket, simply prefix the curly bracket with a pipe symbol: |{ or |}.

Note that since tokens are reprocessed on each iteration (and although certain measures are taken within VoiceAttack to prevent it), <u>you can end up in an infinite loop and/or cause VoiceAttack to crash</u>, so use nested tokens with caution. If you are using an older command and it is giving you problems, you can uncheck the option, ‘Use Nested Tokens’ on the options screen to see if that that helps out.

Note: VoiceAttack's tokens work even when there are multiple entries for random TTS statements. Remember, **tokens are case-sensitive**... {Time} will **not** be processed properly, but, {TIME} or {time} will work just fine.

Below is a haphazard list of tokens in no particular order (mostly in the order they were created… lol… sorry, I will do better at some point):

{TIME} - The current time, expressed in a 24-hour clock.
{TIME:dateVariableName} - The indicated date variable's time, expressed in a 24-hour clock.

{time} - The current time, expressed as a 12-hour clock.
{time:dateVariableName} - The indicated date variable's time, expressed as a 12-hour clock.

{TIMEHOUR} - The hour of the current time, expressed in a 12-hour clock.
{TIMEHOUR:dateVariableName} - The hour of the indicated date variable's time, expressed in a 12-hour clock.

{TIMEHOUR24} - The hour of the current time, expressed in a 24-hour clock.
{TIMEHOUR24:dateVariableName} - The hour of the indicated date variable's time, expressed in a 24-hour clock.

{TIMEMINUTE} - The minute of the current time.
{TIMEMINUTE:dateVariableName} - The minute of the indicated date variable's time.

{TIMESECOND} - The second of the current time.
{TIMESECOND:dateVariableName} - The second of the indicated date variable's time.

{TIMEMILLISECOND} - The second of the current time.

{TIMEMILLISECOND:dateVariableName} - The second of the indicated date variable's time.

{TIMEAMPM} - The AM/PM designation of the current time (or the proper designation of AM/PM in the set culture).
{TIMEAMPM:dateVariableName} - The AM/PM designation of the indicated date variable's time (or the proper designation of AM/PM in the set culture).

{TIMESTAMP} - This renders the current date/time as a four-digit year, two-digit month, two-digit day, two-digit hour (24 hour), two-digit minute, two-digit second and three-digit millisecond (like this: 20200101123020500).
{TIMESTAMP:dateVariableName} - The timestamp of the indicated date variable's date/time.

{DATE} - The current date, formatted as follows: 'April 3, 2020'.
{DATE:dateVariableName} - The indicated date variable, formatted as follows: 'April 3, 2020'.

{DATEYEAR} - The current year as a number (2019, 2020, 2021, etc.).
{DATEYEAR:dateVariableName} - The indicated date variable's year as a number (2019, 2020, 2021, etc.).

{DATEDAY} - The current day of the month as a number.
{DATEDAY:dateVariableName} - The indicated date variable's day of the month as a number.

{DATEMONTH} - The current month, spelled out ('April', 'May', 'June', etc.).
{DATEMONTH:dateVariableName} - The indicated date variable's month, spelled out ('April', 'May', 'June', etc.).

{DATEMONTHNUMERIC} - The current month as a number (April would be 4, May would be 5, June would be 6, etc.).
{DATEMONTHNUMERIC:dateVariableName} - The indicated date variable's month as a number (April would be 4, May would be 5, June would be 6, etc.).

{DATEDAYOFWEEK} - The current day of the week spelled out ('Monday', 'Tuesday', 'Wednesday', etc.).
{DATEDAYOFWEEK:dateVariableName} - The indicated date variable's day of the week spelled out ('Monday', 'Tuesday', 'Wednesday', etc.).

{DATETICKS} - The current date as ticks. This will be a numeric value expressed as text that can be used for comparison.
{DATETICKS:dateVariableName} - The indicated date variable as ticks. This will be a numeric value expressed as text that can be used for comparison.

{DATETIMEFORMAT:textFormatVariable} - The current date/time, formatted using standard format strings. The textFormatVariable is a text variable that is required. For example, {DATETIMEFORMAT:myFormat}, with myFormat set to ‘d MMMM yyyy’ and the current date as the 4th of May, 2020 will render, ‘4 May 2020’. May the fourth be with you ;)

{DATETIMEFORMAT:dateVariableName:textFormatVariable} - The indicated date/time variable, formatted using standard format strings. The textFormatVariable is a text variable that is required. For example, {DATETIMEFORMAT:myDate:myFormat}, with myFormat set to 'd MMMM yyyy' and myDate set as the 4th of May, 2020 will render, '4 May 2020'. May the fourth be with you (always) ;)

{CMD} - The name of the currently-executing command.

{CMDSEGMENT:*segment*} – If your command contains, 'dynamic command sections' (see, 'Dynamic Command Sections' in the, 'Command Screen' documentation above), you can retrieve specific portions of the spoken command, indicated by its numeric position. This will allow you to make more precise decisions based on what was spoken.

For instance, let's say you have a complex dynamic command that you use to build several kinds of items like this: **'**build [1..10][bulldogs;wolves;strikers][please;]'
Then, you say, 'build 5 bulldogs' to execute that command. To find out *what was built*, you can check {CMDSEGMENT:2} (note that the specified, 'segment' is zero-based. So, the first segment is 0. The second is 1, and so on). The rendered value will be, 'bulldogs'. Then, you can find out *how many items to build* by checking {CMDSEGMENT:1}. The rendered value will be, '5' (which you can convert and use in a loop, for instance). For bonus points, you can check {CMDSEGMENT:3} and see if, 'please' was spoken and then thank the user for being so polite (or chide them if they didn't say, 'please') ;)
Note: This token will render as, 'Not set' if called from a command triggered by keyboard, mouse or joystick, and the command is a root command.

{CMDACTION} - The method by which the current command was executed. The possible results are, 'Spoken', 'Keyboard', 'Joystick', 'Mouse', 'Profile', 'External', 'Unrecognized', 'ProfileUnloadChange', 'ProfileUnloadClose', 'DictationRecognized', 'Plugin' and 'Other'. The value will be, 'Spoken' if the command was executed by a spoken phrase, 'Keyboard' if the command was executed using a keyboard shortcut, 'Joystick' if executed by a joystick button, 'Mouse' if executed by a mouse button click and 'Profile' if the command was executed on profile load (from the command indicated in the profile options screen). The value will be 'External' if the command is executed from a command line or if you right-click on a command on the profile screen and execute it from there. 'Unrecognized' will be the value if the command was executed using the unrecognized phrase catch-all command in the profile options screen. 'ProfileUnloadChange' and 'ProfileUnloadClose' will be rendered if the command is executed as part of the profile being unloaded, either by changing the profile or by VoiceAttack shutting down. 'DictationRecognized' is rendered if the command was invoked as a result of a dictation phrase being recognized. A value of, 'Plugin' is rendered if the command is executed from a plugin or inline function. 'Other' is reserved.

{CMD_BEFORE} - When using wildcards in spoken phrases, this is the text that occurs ***before*** the wildcard phrase. For example, if using, '*rocket*' as your wildcard phrase and you say, 'i am going for a ride in my rocket ship' {CMD_BEFORE} will contain, 'i am going for a ride in my' and {CMD_AFTER} will contain 'ship'. This token does not have to be set prior to using.

{CMD_AFTER} - When using wildcards in spoken phrases, this is the text that occurs

after the wildcard phrase.
{CMD_WILDCARDKEY} - When using wildcards in spoken phrases, this is the text that is not variable. For example, if using, '*rocket*' as your wildcard phrase and you say, 'i am going for a ride in my rocket ship' {CMD_WILDCARDKEY} will render as, 'rocket'.

{CMDCONFIDENCE} - This token provides the confidence level that the speech engine provides when the speech engine detects speech. The value range for a spoken command is from “0” to “100”. This value will always be, “0” when a command is not executed as a spoken command (use the, ‘{CMDACTION}’ token to check how the command was executed). Note this value is accessible from within the specified unrecognized catch-all command.

{CMDMINCONFIDENCE} - This token provides the minimum confidence level set by the user as it applies to the executing command. The value range is “0” to “100”. This value will be, “0” if the minimum confidence level is not set. Note that this token can return a value even if the command is not spoken.

{LASTSPOKENCMD} - This provides the last-spoken command phrase. Useful from within sub-commands where you need to know what was said to invoke the root command, or if you need to know what was spoken prior to the current command (if the current command was not spoken (executed by keyboard, mouse, joystick, external, etc.).

{PREVIOUSSPOKENCMD} - This provides the spoken command phrase prior to the last spoken command phrase.

{SPOKENCMD:*value*} - This provides access to the history of spoken commands (up to a maximum of 1000 phrases), starting at 0. A value of zero ({SPOKENCMD:0}) is the same as getting the value of the {LASTSPOKENCMD} token. A value of 1 is the same as getting the value of {PREVIOUSSPOKENCMD} ({SPOKENCMD:1}). A value of 2 would get the spoken phrase that was issued before the previous spoken command and so on. If no history exists for the value, a blank (empty string) is rendered.

{LASTSPOKEN} - This renders the text captured by the speech engine regardless of whether or not the command was recognized. Note that the rendered value will always be lowercase.

{PREVIOUSSPOKEN} - This renders the previous text captured by the speech engine regardless of whether or not the command was recognized. Note that the rendered value will always be lowercase.

{CMDISSUBCOMMAND} - This token will render as ‘1’ if the currently-executing command is executing as a subcommand (a command that is executed by another command). The rendered value will be, ‘0’ if the command is not a subcommand.

{CMDDOUBLETAPINVOKED} - This token will render as ‘1’ if the currently-executing command was executed as the result of a double tap. The rendered value will be, ‘0’ if not.

{CMDLONGPRESSINVOKED} - This token will render as ‘1’ if the currently-executing command was executed as the result of a long press. The rendered value will be, ‘0’ if not.

{CMDCOUNT} - This token provides the count of top-level commands (commands that are not subcommands) that have executed since VoiceAttack had launched.

{CMDWHENISAY} - This token provides the full value of what is indicated in the, 'When I Say' input box on the command screen.

{CMDACTIVE:*name*} - This provides a way to test if a command is currently active. Simply provide the command name (spoken phrase), and the rendered value will be, '1' if the command is active, or '0' if not. Ex: {CMDACTIVE:fire weapon} will check to see if the command, 'fire weapon' is active.

{CMDACTIVECOUNT:*name*} - This will render as the number of active instances of a command (spoken phrase). Ex: {CMDACTIVECOUNT:fire weapon} will check to see how many instances of the command, 'fire weapon' are currently executing.

{CMDEXISTS:*name*} - This provides a way to test if a command exists. Simply provide the command name (spoken phrase), and the rendered value will be, '1' if the command exists, or '0' if not. Ex: {CMDEXISTS:fire weapon} will check to see if the command, 'fire weapon' exists.

{CMDALREADYEXECUTING} - This provides a way to test if the current command is already executing in another instance. The rendered value will be, '1' if the command is already executing, or '0' if not.

{CMDLASTUSEREXEC} – This renders the number of seconds since the last command executed by the user - spoken phrase, keyboard key press, mouse click or joystick button press. That is, it does not include subcommands, commands executed externally, right-click execute, etc.

{ISLISTENINGOVERRIDE} - If the executing command was invoked by a listening override keyword (for instance, if you executed the command by saying, 'Computer, Open Door' instead of, 'Open Door') the result will be, '1'. Otherwise, the result will be, '0'.

{ISCOMPOSITE} - If an executing command is composite (where there is a prefix and a suffix), the return of this token will be, '1'. Otherwise, it will be, '0'.

{PREFIX} - If an executing command is composite (where there is a prefix and a suffix), this token will be replaced with the prefix portion of the command. If called in a non-composite command, this will result in a blank string.

{SUFFIX} - If an executing command is composite (where there is a prefix and a suffix), this token will be replaced with the suffix portion of the command. If called in a non-composite command, this will result in a blank string.

{COMPOSITEGROUP} - If an executing command is composite (where there is a prefix and a suffix), this token will return the group value if it is being used.

{CATEGORY} - This returns the category of the executing command. If this token is used

on a composite command (a command using a prefix and suffix), the result will be the category of the prefix and the category of the suffix separated by a space. For the individual parts of a composite category, see, ‘{PREFIX_CATEGORY}' and '{SUFFIX_CATEGORY}'.

{PREFIX_CATEGORY} - This returns the prefix category of the executing command (if the executing command is a composite command).

{SUFFIX_CATEGORY} - This returns the suffix category of the executing command (if the executing command is a composite command).

{QUEUESTATUS:*name*} - This renders the status of a command execution queue indicated in the *name* parameter. The following values will be rendered:

‘Not Initialized’ – This will be the rendered status if the queue does not exist (that is, no commands have been enqueued into a queue by the name given).
‘Running’ - The queue is currently running (not paused) and there is at least one command in the queue.
‘Idle’ - The queue is running (not paused) but there are zero items in the queue.
‘Paused’ - The queue is paused, or flagged to be paused if a command is currently executing within the queue.
‘Stopped’ - The queue is in a stopped state or has been flagged to stop if a command is currently executing within the queue.

{QUEUECMDCOUNT:*name*} - This renders the number of commands contained within a command execution queue that is indicated in the *name* parameter. If the queue does not exist, ‘0’ will be rendered.

{QUEUEACTIVECMD:*name*} - This renders the name of the executing command of a command execution queue (indicated in the *name* parameter). If the queue does not exist, or if there is no command currently executing within the queue, an empty value ‘’ will be rendered.

{QUEUECOUNT} - This renders the number of command execution queues that are currently available.

{PROFILE} - This renders the name of the currently-loaded profile.

{PROFILE_AT1}, {PROFILE_AT2}, {PROFILE_AT3} - These render the three different AuthorTags of the currently-loaded profile (see, ‘VoiceAttack Author Flags’ later on in this document). This is an advanced feature that will probably not be used by most.

{PREVIOUSPROFILE} - This provides the name of the profile that was loaded prior to the currently-loaded profile.

{PREVIOUSPROFILE_AT1}, {PREVIOUSPROFILE_AT2}, {PREVIOUSPROFILE_AT3} - These render the three different AuthorTags of the profile that was loaded prior to the currently-loaded profile (see, ‘VoiceAttack Author Flags’ later on in this document). This is an advanced feature that will probably not be used by most.

{PROFILE:*value*} - This provides access to the history of the names of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero ({PROFILE:0}) is the same as getting the value of the {PROFILE} token. A value of 1 is the same as getting the value of {PREVIOUSPROFILE} ({PROFILE:1}). A value of 2 would get the name of the profile loaded before the previous profile and so on. If no history exists for the value, a blank (empty string) is rendered.

{PROFILE_AT1:*value*}, {PROFILE_AT2:*value*}, {PROFILE_AT3:*value*} - This provides access to the history of the three AuthorTag values of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero ({PROFILE_AT1:0}) is the same as getting the value of the {PROFILE_AT1} token. A value of 1 is the same as getting the value of {PREVIOUSPROFILE_AT1} ({PROFILE_AT1:1}). A value of 2 would get the Author Tags of the profile loaded before the previous profile and so on. If no history exists for the value, a blank (empty string) is rendered. (See, 'VoiceAttack Author Flags' later on in this document). This is an advanced feature that will probably not be used by most.

{NEXTPROFILE} - This provides the name of the profile that has been selected to load after the current profile is unloaded. This token will render as empty if the profile is not unloading or if the profile is unloading due to VA shutting down. Note: If you haven't been able to tell by now, this token is only available from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document).

{NEXTPROFILE_AT1}, {NEXTPROFILE_AT2}, {NEXTPROFILE_AT3} - These provide the three different AuthorTag values of the profile that has been selected to load after the current profile is unloaded. These tokens will render as empty if the profile is not unloading or if the profile is unloading due to VA shutting down. Note: Just like {NEXTPROFILE}, These tokens are only available from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document, as well as the, 'Author Flags' section later in this document). This is an advanced feature that will probably not be used by most.

{RANDOM:*lowValue*:*highValue*} - A random integer between a low value and a high value (inclusive) will be generated. For example, {RANDOM:1:100} will pick a number between 1 and 100. {RANDOM:77:199} will pick a number between 77 and 199.

{RANDOMDEC:*lowValue*:*highValue*} - A random decimal between a low value and a high value (inclusive) will be generated. The number of decimal places provided will be determined by the values provided in lowValue or highValue. For example, {RANDOM:0.2:5.4444} will pick a number between 0.2 and 5.4444. An example of a returned value from that range could be 3.1234. Note this has four decimal places, since highValue is using four decimal places.

***Important Note: <u>Conditions have been renamed to Small Integer</u>**. The token {SMALL:variableName} (see below) has been created to handle small integer variables. The {COND:conditionName} and {CONDFORMAT:conditionName} have been left in for backward compatibility (you can use either token interchangeably, since they both access the same values).

{SMALL:*variableName*} - The numeric value stored in a small integer variable will be retrieved. For example, if you have a variable called, 'My Small Int', you would use, {SMALL:My Small Int}. The referenced variable must be set prior to using, otherwise the value will be, 'Not set' when accessed. NOTE: The comma formatting of this token has been removed. See, '{SMALLFORMAT}' token, below).

{SMALL:*variableName:defaultValue*} - The small integer value stored in a small.l integer variable will be retrieved (just like above), but, if the value results in, 'Not set', the value in the, 'defaultValue' parameter will be used. This referenced variable does not have to be set prior to using (since the default value will be used instead).

{SMALLFORMAT:*variableName*} - This returns the same value as {SMALL}, but the value is formatted with commas (for TTS). This used to be the default behavior of {SMALL}, but, since the introduction of the {EXP} token, this token had to be created.

{SMALLFORMAT:*variableName:defaultValue*} - This returns the same value as {SMALLFORMAT}, except if the variable is, 'Not set', the value indicated as the default will be used.

{INT:*variableName*} - The numeric value stored in an integer variable will be retrieved. For example, if you have a variable called, 'My Int', you would use, {INT:My Int}. The referenced variable must be set prior to using, otherwise the value will be, 'Not set' when accessed. NOTE: The comma formatting of this token has been removed. See, '{INTFORMAT}' token, below).

{INT:*variableName:defaultValue*} - The integer value stored in an integer variable will be retrieved (just like above), but, if the value results in, 'Not set', the value in the, 'defaultValue' parameter will be used. This referenced variable does not have to be set prior to using (since the default value will be used instead).

{INTFORMAT:*variableName*} - This returns the same value as {INT}, but the value is formatted with commas (for TTS). This used to be the default behavior of {INT}, but, since the introduction of the {EXP} token, this token had to be created.

{INTFORMAT:*variableName:defaultValue*} - This returns the same value as {INTFORMAT}, except if the variable is, 'Not set', the value indicated as the default will be used.

{DEC:*variableName*} - The numeric value stored in a decimal variable will be retrieved. For example, if you have a variable called, 'My Decimal', you would use, {DEC:My Decimal}. The referenced variable must be set prior to using, otherwise the value will be, 'Not set' when accessed. **NOTE**: If you plan on using this token for calculations (for example, with the {EXP} token), or, if you plan on sharing your profile with folks in other countries, you will want to use the {DECINV} token instead (see below).

{DEC:*variableName:defaultValue*} - The decimal value stored in a decimal variable will be retrieved (just like above), but, if the value results in, 'Not set', the value in the, 'defaultValue' parameter will be used. This referenced variable does not have to be set

prior to using (since the default value will be used instead). **NOTE**: If you plan on using this token for calculations (for example, with the {EXP} token), or, if you plan on sharing your profile with folks in other countries, you will want to use the {DECINV} token instead (see below).

{DECINV:*variableName*} and {DECINV:*variableName:defaultValue*} - This works exactly the same as the {DEC} tokens above, except that the value will be rendered using the invariant culture. That means that your decimal value will always be rendered with a point ('.') instead of a comma, (','). You will want to use this token when needing decimal variables to be used in calculations (such as with {EXP} and {EXPDECINV}) and if you intend on sharing your profiles with others in different countries.

{BOOL:*variableName*} - The text representation of the boolean value will be retrieved. If the variable value is true, the token will be replaced with, 'True'. If false, the token will be replaced with, 'False'. The referenced variable must be set prior to using, otherwise the rendered value will be, 'Not set' when accessed.

{BOOL:*variableName:defaultValue*} - The Boolean (true/false) value stored in a Boolean variable will be retrieved (just like above), but, if the value results in, 'Not set', the value in the, 'defaultValue' parameter will be used. This referenced variable does not have to be set prior to using (since the default value will be used instead).

{TXT:*variableName*} - The text value stored in a text variable will be retrieved. This works the same way as all the other types (above), except that... well... you get the idea. The variable referenced by this token must be set prior to using, otherwise the rendered value will be, 'Not set' when accessed.

{TXT:*variableName:defaultValue*} - The text value stored in a text variable will be retrieved (just like above), but, if the value results in, 'Not set', the text in the, 'defaultValue' parameter will be used. This referenced variable does not have to be set prior to using (since the default value will be used instead).

{TXTURL:*variableName*} - The text value stored in a text variable will be retrieved (again, just like above), but the result will be URL encoded.

{TXTRANDOM:*value*} - This token will do a simple randomize of its text contents. For example, a token could be used in TTS to say, 'I love {TXTRANDOM:food;animals;jewelry}' and the result would be, 'I love animals' or, 'I love food' or, 'I love jewelry'. Note that the, 'value' portion can be another token, however, if you want to randomize randomized tokens, it is suggested that you use text value variables to store the values, otherwise it may get a little out of sorts. Experiment with this one, since the permutations for testing are a little out there :)

{TXTLEN:*variableName / value*} - This will return the length of the text variable's value. This token can also evaluate a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, 'myTextVariable' is set to, 'weapons', {TXTLEN:myTextVariable} will evaluate to "7". {TXTLEN:"{TXT:myTextVariable}"} will also evaluate to "7". {TXTLEN:"apple pie"} will evaluate to "9".

{TXTUPPER:*variableName / value*} - This will return the text variable's value in upper case. This token can also render a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextUpper’ is set to, ‘citadel’, {TXTUPPER:myTextUpper} will render as “CITADEL”. {TXTUPPER:”{TXT:myTextUpper}”} will also render as, “CITADEL”. {TXTUPPER:”apple pie”} will render as, “APPLE PIE”.

{TXTLOWER:*variableName / value*} - This will return the text variable's value in lower case. This token can also render a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextLower’ is set to, ‘TeXaS’, {TXTLOWER:myTextLower} will render as “texas”. {TXTLOWER:”{TXT:myTextLower}”} will also render as, “texas”. {TXTLOWER:”PECAN PIE”} will render as, “pecan pie”.

{TXTTRIM:*variableName / value*} - This will return the text variable's value with spaces removed from the beginning and end. This token can also process a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextTrim’ is set to, ‘ Parachute ’, {TXTTRIM:myTextTrim} will return as “Parachute”. {TXTTRIM:”{TXT:myTextTrim}”} will also return as, “Parachute”. {TXTTRIM:”Yellow Submarine ”} will return as, “Yellow Submarine”.

{TXTREPLACEVAR:*variableSource / value:variableFrom / value:variableTo / value*} - This will render the text variable's value with the text indicated in *variableFrom* as the value in *variableTo*. For example: Variable, 'myVariable' value set to, 'This is a test'. Variable, ‘mySearch’ value set to, ‘test’ and variable, ‘myReplacement’ value set to, ‘monkey’. {TXTREPLACEVAR:myVariable:mySearch:myReplacement} renders as, ‘This is a monkey’. This token can also process a literal text value or token for each parameter, if each literal text or token is contained **between double quotes**. For example, {TXTREPLACEVAR:myVariable:”test”:”monkey”} will also render as, “This is a monkey”.

{TXTREGEXREPLACE:*variableSource:variableFrom / value:variableTo*} -This will return the text variable's value with the text matched as a regular expression in *variableFrom* as the value in *variableTo*. For example: Variable, 'myVariable' value set to, 'a. b- c+ d^'. Variable, ‘myRegexMatch’ value set to, ‘[^a-zA-Z0-9]’ (this is a regular expression to match any non-alphanumeric characters, except spaces (without quotes – note that there is a space after, ‘9’)) and variable, ‘myReplacement’ value set to, ‘#’ (without quotes). {TXTREGEXREPLACE:myVariable:mySearch:myReplacement} will render as, ‘a# b# c# d#’ (all of the special characters (except spaces) are replaced by, ‘#’).

{TXTNUM:*variableName / value*} - This will attempt to remove all characters except numeric (0-9, ., -). This token can also render a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextNum’ is set to, ‘8675309 Jenny’, {TXTNUM:myTextNum} will return as “8675309”. {TXTNUM:”{TXT:myTextNum}”} will also render as, “8675309”. {TXTNUM:”Brick, 08724”} will render as, “08724”.

{TXTALPHA:*variableName / value*} - This will attempt to remove all numeric characters (0-9). This token can also render a literal text value or token, if the literal text or token is

contained **between double quotes**. For example, if text variable, ‘myTextAlpha’ is set to, ‘8675309 Jenny’, {TXTALPHA:myTextAlpha} will return as “ Jenny”. {TXTALPHA:”{TXT:myTextNum}”} will also render as, “ Jenny”. {TXTALPHA:”Brick, 08724”} will render as, “Brick, ”.

{TXTWORDTONUM:*variableName / value*:selection / value} - This will replace numeric words with integer representations. This is to assist (workaround) in the way that the speech engine interprets single-digit numbers. For instance, the speech engine interprets, “one” as, “one”, “two” as “two”, but it interprets, “ten” as “10”, “eleven” as “11” and so on. The *variableName* parameter is the parameter that will hold the text to convert. The *variableName* parameter can accept a variable name as well as a token/literal as long as the token/literal is **between double quotes**. The *selection* parameter is optional. It can also be a variable name or token/literal if the token/literal is between double quotes. The *selection* parameter allows you to specify something other than English word replacements. The value for *selection* must be a comma-delimited list of exactly ten items (starting at “zero”) in order to work. Here are some examples:
{TXTWORDTONUM:”One thing leads two another”} will render as “1 thing leads 2 another”. Note that the second parameter is omitted, since the default is English.

In this next example, myText variable value is set to “viajando al sector uno”. {TXTWORDTONUM:myTextVariable:“cero,uno,dos,tres,quatro,cinco,seis,siete,ocho,nueve”} will render as “viajando al sector 1”. Note that the *selection* parameter is a literal value that is enclosed in double quotes, has ten items and is comma-delimited.

{TXTTITLE:*variableName / value*} - This will attempt to title case the value (for supported locales). For example, 'war and peace' becomes, 'War And Peace'. Yes… I know that's not *proper* title case (at least in English), but it's as close as the framework will allow without linguistic rules. Sorry!
This token can also render a literal text value or token, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextTitle’ is set to, ‘napoleon dynamite’, {TXTTITLE:myTextTitle} will return as “Napoleon Dynamite”. {TXTTITLE:”{TXT:myTextTitle}”} will also render as, “Napoleon Dynamite”. {TXTTITLE:”nacho libre”} will render as, “Nacho Libre”.

{TXTCONCAT:*variableName1 / value:variableName2 / value*} - This will attempt to concatenate the text in variable 2 to the text of variable 1. For instance, let's say var1 is set to “Good” and var2 is set to “ Morning” (note the space). {TXTCONCAT:var1:var2} will yield, “Good Morning”. If variable 1 or variable 2 are not set, the rendered result will be, “Not set”. This token can also render a literal text value or token for either parameter, if the literal text or token is contained **between double quotes**. For example, if text variable, ‘myTextConcat’ is set to, ‘Good’, {TXTCONCAT:myTextConcat, “ Morning”} will return as “Good Morning”. {TXTCONCAT:”Good” : “ Morning”} will also render as, “Good Morning”.

{TXTSUBSTR:textVariableOf / text value:intVariableBegin / int value : intVariableLength / int value} - this will return a portion (substring) of the text provided in textVariableOf, starting at intVariableBegin with a length of intVariableLength. This token uses three parameters that may contain variables, literal values or tokens. If variables are used in any position, the variables **MUST BE SET** prior to using. If a literal value is used in

the textVariableOf/text value parameter, make sure that the literal value is **between double quotes**. The value indicated for the beginning of the substring (second parameter, intVariableBegin) is zero-based. That means, if you want to retrieve the substring at the beginning of the provided text (txtVariableOf), set the beginning to zero. If the value of intVariableBegin resolves to, 'Not set', zero will be assumed. To indicate the length of the substring, provide the length in the third parameter (intVariableLength variable). If the value in the third parameter resolves to, 'Not set' or if the value in the third parameter ends up past the end of the provided text, the end of the provided text is assumed. If either the value of intVariableBegin or intVariableLength is less than zero, the result will be an empty (blank) string.

{TXTSUBSTR:"We are the Champions":11:9} will render as, 'Champions' (note that, 'Champions' starts at position 12, but since the beginning position is zero-based, the value is 11).

{TXTPOS:textVariableOf / text value:textVariableIn / text value : intVariableStart / int value} - this will return the position of text (txtVariableOf) within more text (txtVarIn) starting at a given point (intVarStart). This uses three parameters that may contain variables, literal values or tokens. If variables are used in any position, the variables **MUST BE SET** prior to using. The text search is case-sensitive. If the text is found, the result will be the zero-based index of the start of the text. Otherwise, if the text is not found the result will be "-1".

For example (**using ALL VARIABLES**), if you are wanting to find the word, 'active' in the phrase, 'all systems active' you will first need to set a text variable to the word, 'active'. In this example, that variable is called, 'txtFindActive'. Next, you will need to set a second text variable to the phrase, 'all systems active'. That variable is called, 'txtStatus'. Next, you will need to set a variable to indicate the position where you would like to start searching. Since you would like to search the entire phrase, you will need to set this variable to zero. So, a third variable, 'intPosition' needs to be set to zero (see note on this below). You will then use this token with the three variables like this:
{TXTPOS:txtFindActive:txtStatus:intPosition}. The result will be "12", as the position of 'active' appears in position 12 (the phrase actually starts at the 13th character, but we are using a zero-based positioning system).
Note: if txtVariableOf or txtVariableIn of are not set, the result will be, '-1'. If intVarStart is not set, 0 is assumed (for convenience).

Another example, using variables AND literal values based on the example above, you can create a statement that does the same thing: {TXTPOS:"active":txtStatus:5}. This will search for the word, 'active' starting at position 5 (it still returns, "12"). Note that **double quotes are required** if you are using literal text in the first and second parameters (and not required for the last parameter).

{TXTLASTPOS:textVariableOf / text value:textVariableIn / text value}
and
{TXTLASTPOS:textVariableOf / text value:textVariableIn / text value : intVariableStart / int value} - this will return the last position of text (txtVariableOf) within more text (txtVarIn) starting at a given point (intVarStart). This uses three parameters that may contain variables, literal values or tokens. If variables are used in any position, the variables **MUST BE SET** prior to using. If literal values are used in any position, make sure

the literal values are **between double quotes**. The text search is case-sensitive. If the text is found, the result will be the zero-based index of the start of the text. Otherwise, if the text is not found the result will be “-1”. One thing you will need to understand about this token is that, unlike {TXTPOS:}, the start position is from the end of the text to the beginning. So, the search will begin at the length of the text and move to zero. For example, {TXTLASTPOS:”abc”, “123abc”:0} will return “-1” because the search begins and ends at 0 (the beginning). {TXTLASTPOS:”abc”, “abc123abc”:9} will return “6” because the search begins at 9 (the end of the text) and moves left. Note that start positions that are beyond the length of the search string will be adjusted to the length of the search string.

Note that there are two variations of this token. One with a numeric start position and one without. If you use the option without a start position, the end of the text is assumed. {TXTLASTPOS:”abc”, “abc123abc”} will again return “6”.

For brevity, the variable usage examples for this token are the same as they are for {TXTPOS:}, only that the search begins at the end of the search string instead of the start.

{TXTOCCURRENCES:textVariableOf / text value:textVariableIn / text value} – this will return the number of times a text value appears in another text value. This uses two parameters that may contain variables, literal values or tokens. If variables are used in any position, the variables **MUST BE SET** prior to using. The text search is case-sensitive. If the text is found, the result will be a text representation of a numeric value of the number of times the text is found (for instance, if the text is found five times, “5” will be rendered). Otherwise, if the text is not found the result will be “0”.

For example (**using ALL VARIABLES**), if you are wanting to find out how many times the letter, ‘e’ appears in the phrase, 'all systems active' you will first need to set a text variable to the letter, 'e'. In this example, that variable is called, 'txtFind'. Next, you will need to set a second text variable to the phrase, 'all systems active'. That variable is called, 'txtSomePhrase'. You will then use this token with the two variables like this: {TXTOCCURRENCES:txtFind:txtSomePhrase}. The result will be “2”, as the letter, ‘e’ appears twice in 'all syst**e**ms activ**e**'. Note: if either of the variables are not set, the result will be, '0.

Another example, using variables AND literal values based on the example above, you can create a statement that does the same thing: {TXTOCCURRENCES:”e”:txtSomePhrase}. This will search for the literal letter, ‘e’ in variable, ‘txtSomePhrase. Note that **double quotes are required** if you are using literal text in either of the parameters

{SPACE} - Renders as a single space. This really only useful in very specific circumstances when a literal value will not do.

{NEWLINE} - Renders as a new line. Again, this really only useful in very specific circumstances when a literal value will not do.

{GUID} - Renders as a GUID (36-character unique identifier).

{GUIDCLEAN} - Renders as a GUID without the dashes (32-character unique identifier).

{ACTIVEWINDOWTITLE} - Returns the active window's title text.

{ACTIVEWINDOWPROCESSNAME} - Returns the active window's process name (the one you see in Task Manager).

{ACTIVEWINDOWPROCESSID} - Returns the active window's process id (the one you see in Task Manager details).

{ACTIVEWINDOWPATH} - Returns the path of the active window's executable.

{ACTIVEWINDOWWIDTH} - Returns the active window's width. Helps with resizing/moving.

{ACTIVEWINDOWHEIGHT} - Returns the active window's height. Helps with resizing/moving.

{ACTIVEWINDOWTOP} - Returns the active window's top (Y coordinate). Helps with resizing/moving.

{ACTIVEWINDOWLEFT} - Returns the active window's left (X coordinate). Helps with resizing/moving.

{ACTIVEWINDOWRIGHT} - Returns the active window's right (left + width). Helps with resizing/moving.

{ACTIVEWINDOWBOTTOM} - Returns the active window's bottom (top + height). Helps with resizing/moving.

{PROCESSEXISTS:*textVariable*} – Returns “1” if a process with the name specified in *textVariable* exists. Returns “0” if not. Note that this can take wildcards (*). For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

{PROCESSCOUNT:*textVariable*} – Returns “1” or more if processes with the name specified in *textVariable* exist. Returns “0” if there are none. Note that this can take wildcards (*). For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for processes that have a name that contains 'notepad'.

{PROCESSFOREGROUND:*textVariable*} – Returns “1” if a process with a main window with the title specified in *textVariable* is the foreground window. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

{PROCESSMINIMIZED:*textVariable*} – Returns “1” if a process with a main window with the title specified in *textVariable* is minimized. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

{PROCESSMAXIMIZED:*textVariable*} – Returns “1” if a process with a main window with the title specified in *textVariable* is maximized. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

{WINDOWEXISTS:*textVariable*} – Returns “1” if a window with the title specified in *textVariable* exists. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

{WINDOWFOREGROUND:*textVariable*} – Returns “1” if a window with the title specified in *textVariable* is the foreground window. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

{WINDOWMINIMIZED:*textVariable*} – Returns “1” if a window with the title specified in *textVariable* is minimized. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

{WINDOWMAXIMIZED:*textVariable*} – Returns “1” if a window with the title specified in *textVariable* is maximized. Returns “0” if not. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

{WINDOWCOUNT:*textVariable*} – Returns “1” or more if windows with the title specified in *textVariable* exist. Returns “0” if there are none. Note that this can take wildcards (*) for window titles that change. For instance, if *textVariable* contains '*notepad*' (without quotes), the search will look for windows that have a title that contains 'notepad'.

{WINDOWTITLEUNDERMOUSE} – Returns the title for the window that is currently located under the mouse.

{WINDOWPROCESSUNDERMOUSE} – Returns the process name for the window that is currently located under the mouse.

{CMDTARGETFOREGROUND} – Renders “1” if the current command’s target is the foreground window. Renders “0” if the target is not the foreground window or if the target does not exist.

{CMDTARGETMINIMIZED} – Renders “1” if the current command’s target is minimized. Renders “0” if the target is not minimized or if the target does not exist.

{CMDTARGETMAXIMIZED} – Renders “1” if the current command’s target is maximized. Renders “0” if the target is not maximized or if the target does not exist.

{MOUSESCREENX} - Returns the X coordinate of the mouse position as it relates to the screen. Zero would be the top-left corner of the screen.

{MOUSESCREENY} - Returns the Y coordinate of the mouse position as it relates to the screen. Zero would be the top-left corner of the screen.

{MOUSEWINDOWX} - Returns the X coordinate of the mouse position as it relates to the active window. Zero would be the top-left corner of the window.

{MOUSEWINDOWY} - Returns the Y coordinate of the mouse position as it relates to the active window. Zero would be the top-left corner of the window.

{CAPSLOCKON} - Returns “1” if the caps lock key is locked. Returns “0” if not.

{NUMLOCKON} - Returns “1” if the numlock key is locked. Returns “0” if not.

{SCROLLLOCKON} - Returns “1” if the scroll lock key is locked. Returns “0” if not.

{CLIP} - This token will be replaced by whatever is in the Windows clipboard, as long as what is in the clipboard is a text value. **Note**: This value can also be set within VoiceAttack by using the 'Set a text value to the Windows clipboard' action.

{DICTATION} - This token will be replaced by whatever is in the dictation buffer, without any formatting. So, if you said, 'This is a test of the emergency broadcast system' and then later said 'this is only a test', the result would be 'This is a test of the emergency broadcast system this is only a test'.

{DICTATIONON} - This token returns “1” if dictation mode is on, and “0” if it is off.

{DICTATION:*options*} - This token is an attempt to offer some basic touch up to whatever is in the dictation buffer. The speech recognition engine may do all kinds of heinous things to your text (as you will see... lol ;)) First, let’s talk about the options. The option examples below will be used with as if your dictation buffer was filled first by saying 'This is a test of the emergency broadcast system' and then later saying 'this is only a test'.

The options are:

PERIOD - This puts a period in for you at the end of each line (so you don’t have to constantly say, 'period' to the speech engine). The rendered output would be:
'This is a test of the emergency broadcast system. this is only a test.'.

CAPITAL - Capitalizes the first character of every line (since the speech engine may or may not do it for you... o_O). The rendered output would be:
'This is a test of the emergency broadcast system This is only a test'.

LATEST - Only display the last thing you said. In the example, you would get 'this is only a test'.

UPPERCASE - Converts all characters to upper case:
'THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM THIS IS ONLY A TEST'

LOWERCASE - Converts all characters to lower case:

'this is a test of the emergency broadcast system this is only a test'

NEWLINE - Makes sure each phrase is on its own line (note that it won't show up this way if you write to the log... the log doesn't care about new lines). Rendered output would be:
'This is a test of the emergency broadcast system
 this is only a test.'.

SPACEX - Replace the, 'X' with an integer number, and that many extra spaces will be placed at the end of each phrase. The default is one space. So, if you use {DICTATION:SPACE4}, the rendered output will be
'this is a test of the emergency broadcast system this is only a test'.

You can use any or all of the options, in any order, separated by a colon. They are not case-sensitive. So, if you simply wanted to add a period to the end of each spoken dictation phrase and ensure that the first character is always capitalized, you would use {DICTATION:PERIOD:CAPITAL}. The above example would be rendered as:
'This is a test of the emergency broadcast system. This is only a test.'.
If you wanted to also make it so that a line feed is placed between lines,
you can use {DICTATION:PERIOD:CAPITAL:NEWLINE}. You will then get:
'This is a test of the emergency broadcast system.
 This is only a test.'.

{EXP:*expression*} and {EXPDECINV:expression} –

UPDATE: If you plan on using the {EXP} token for decimal calculations, it is now recommended that you use {EXPDECINV} instead, especially if you plan on sharing your profile with others in different countries. The {EXPDECINV} token works exactly the same as the {EXP} token, except that the value will be rendered using the invariant culture. That means that your resulting decimal value will always be rendered with a point ('.') instead of a comma, (',').

Currently experimental, the expression token will evaluate a typed-out expression (parenthesis and all, for order of operation) and return the value as text. Comparison expressions can be used on both text and numeric values (including numeric variables such as Integer, Small Integer and Decimal). Arithmetic expressions can be performed on numeric values. Values intended to be used for numeric variables can then be converted using their various, 'Convert Text/Token' option in each of the set value screens (for use in condition blocks (note that the result values can be decimal values when dividing).

IMPORTANT: {EXP} and {EXPDECINV} only accept decimal values expressed in what is called the, 'invariant culture'. That means that it will only accept decimal values using a point ('.') instead of a comma (','). So, if two and a half is expressed as, '2,5', you will need to enter the value as, '2.5' or your result will end up as an expression error.

There are several ways to use expressions as outlined below.

Evaluating Arithmetic Expressions
The expression token can be used to evaluate arithmetic expressions like so:
{EXP: ((5 + 5) - 2) * 10} (evaluates to, '80')

Accepted arithmetic operators: +, -, * (multiplication), / (division), % (modulus).

This token also **accepts tokens** to be evaluated. So, if you have a text value myText1 and it is set to '100' and an integer variable myInt1 that is 200, you can have a mixed expression of, {EXP:({TXT:myText1} + {INT:myInt1}) * 2} that results in '600'.

Numeric Comparisons
You can do numeric comparisons using expressions as well:
{EXP: {TXT:myText1} = {INT:myInt1} } (evaluates to, '0' (false))

Comparison expressions return either '1' for true, or '0' for false.

Accepted comparison operators: =, <, >, <= (less than or equal), >= (greater than or equal), <> (not equal).

You may also use, 'And', 'Or', 'Not':
{EXP: ('cat' = 'dog') And Not (20 = 30) } (evaluates to, '0' (false))
{EXP: ('cat' = 'dog') Or (20 = 30) } (evaluates to, '0' (false))
{EXP: Not ('cat' = 'dog') } (evaluates to, '1' (true))

Evaluating Text Comparisons
The expression token can evaluate text.

{EXP: 'all your base are belong to us' = 'hello'} (evaluates to, '0' (false)).

{EXP: 'all your base are belong to us' <> 'hello'} (evaluates to, '1' (true)).

Just like numeric comparisons, comparison expressions return either '1' for true, or '0' for false.

Note that you have to enclose the text in single quotes (the quotes even need to be around tokens if the token value itself does not have quotes). If the text contains single quotes, they need to be doubled up to be used in expression tokens:
{EXP:'catcher''s mitt' = 'pitcher''s mitt'}

Text comparisons are not case sensitive.

Accepted comparison operators: =, <, >, <= (less than or equal), >= (greater than or equal), <> (not equal). You may also use, 'And', 'Or', 'Not'.

You can use, 'LIKE' as part of your text comparison expressions. The text that you are comparing to needs to have asterisks in various places in order to indicate the type of comparison to make (wildcards).

Asterisks around the text indicate, 'contains':
{EXP: 'rocket ship' LIKE '*rocket*'} (evaluates to '1' (true) because the text contains, 'rocket')

Asterisks at the end indicate, 'starts with' :
{EXP: 'rocket ship' LIKE 'ship*'} (evaluates to '0' (false), since the text does not start with, 'ship')

Asterisks at the beginning indicate, 'ends with':
{EXP: 'rocket ship' LIKE '*ship'} (evaluates to '1' (true), since the text ends with 'ship')

No asterisks indicate exact match (just like using, '=').

Other Expression Abilities
You can concatenate text by using, '+':
{EXP: 'welcome' + ' ' + 'captain' } evaluates to 'welcome captain'.

You can use, 'TRIM', 'SUBSTRING', 'LEN' and, 'IIF' (immediate, 'if')).

TRIM removes any blank spaces around text:
{EXP: TRIM(' my trimmed text   ') } evaluates to 'my trimmed text'

SUBSTRING(*start position*, *length*) gets just the portion of the text you are wanting:
{EXP: SUBSTRING('let the good times roll', 9, 4) } evaluates to, 'good'.
Note that the start is 1-based (not zero-based).

LEN gives the length of the text:
{EXP: LEN('let the good times roll') } evaluates to 23.

IIF(*expression, true part, false part*) allows you to evaluate an expression and get one of two values depending on the result:
{EXP: IIF('cat' = 'dog', 10, 20) } evaluates to, '20'
{EXP: IIF('cat' <> 'dog', 10, 20) } evaluates to, '10'
myCondition1 is 100 and myCondition2 is 200 :
{EXP: IIF({INT:myInt1} > {INT:myInt2}, 'Blue', 'Green') } evaluates to, 'Green'.

VoiceAttack State Tokens

State tokens will allow you to test various conditions that are occurring within VoiceAttack. These can be handy by allowing you to modify the flow of your command actions based on the state of your devices. For instance, you can check (with a conditional (if) statement) if the right mouse button is down when I say, 'fire weapons', missiles are fired instead of photons (lol). Also, if the 'x' button is down, fire both photons and missiles. You can also monitor the position of your sliders or X, Y and Z positions of the joystick itself to create some interesting, 'triggers' using loops and conditional (if) statements.

{STATE_KEYSTATE:*key*} - This token checks to see if a particular key is pressed down or

not. The, 'key' parameter can be any key you can type in a token: 'A', 'B', 'C', 'ß', 'ö', 'ñ', 'ç', as well as keys you can't type in a token: ENTER, TAB, LCTRL, ARROWR (see section later in this document titled, **'Key State Token Parameter Values'** for the full list).

So, if you want to test to see if the F10 key is down, just use the following token: {STATE_KEYSTATE:F10}. To test for the letter 'A', just use this token: {STATE_KEYSTATE:A}. If a keyboard key is down, the replaced value of the token will be "1". If the key is not down, the value will be "0".

{STATE_ANYKEYDOWN} - This token checks to see if any keyboard key is currently pressed. If any keyboard key is down, the replaced value of the token will be "1". If there are no keys pressed down, the value will be "0".

{STATE_LEFTMOUSEBUTTON}
{STATE_RIGHTMOUSEBUTTON}
{STATE_MIDDLEMOUSEBUTTON}
{STATE_FORWARDMOUSEBUTTON}
{STATE_BACKMOUSEBUTTON} - Each of these tokens test to see if a mouse button is being pressed. If you want to test for the right mouse button, use token {STATE_RIGHTMOUSEBUTTON}. If the mouse button is pressed down, the replaced value will be "1". If the mouse button is not pressed, the value will be "0".
Note: If “Do not allow button event to be passed through” is selected on the Mouse Shortcut screen, the rendered value will be “0” for the corresponding button.

{STATE_ANYMOUSEDOWN} - This token checks to see if any of the five standard mouse buttons are currently pressed. If any mouse button is down, the replaced value of the token will be "1". If there are no mouse buttons pressed down, the value will be "0".
Note: If “Do not allow button event to be passed through” is selected on the Mouse Shortcut screen, the rendered value will be “0” if the corresponding button is the only button pressed.

{STATE_MOUSESHORTCUTS} - This token tests to see if VoiceAttack's mouse button shortcuts are enabled. If they are enabled, the replaced value will be "1". Otherwise, the replaced value will be "0".

{STATE_LISTENING} - This token tests to see if VoiceAttack is, 'listening'. If, 'listening' is on, the replaced value will be "1". If, 'listening' is off, the replaced value will be "0".

{STATE_SHORTCUTS} - This token tests to see if VoiceAttack's keyboard shortcuts are enabled. If shortcuts are enabled, the replaced value will be "1". Otherwise, the replaced value will be "0".

{STATE_JOYSTICKSHORTCUTS} - This token tests to see if VoiceAttack's joystick button shortcuts are enabled. If shortcuts are enabled, the replaced value will be "1". Otherwise, the replaced value will be "0".

{STATE_CPU:coreNumber}
{STATE_CPU} - These will return your cpu usage. {STATE_CPU} will return the average for all cores. The value returned will be from “0” to “100”. {STATE_CPU:coreNumber} will

allow you to specify a particular core. For instance, {STATE_CPU:5} will get the cpu usage for core 5.

{STATE_RAMTOTAL} - This will return the total RAM on your system in bytes.

{STATE_RAMAVAILABLE} - This will return the available RAM on your system in bytes.
{STATE_FILEEXISTS:*textVariable*} - This will return “1” if the file indicated in the text variable exists, or “0” if it does not.

{STATE_DIRECTORYEXISTS:*textVariable*} - This will return “1” if the directory indicated in the text variable exists, or “0” if it does not.

{STATE_DIRECTORYHASFILES:*textVariable*} - This will return “1” if the directory indicated in the text variable has files in it, or “0” if it does not. Note that if the directory does not exist, “0” will be returned.

{STATE_AUDIOLEVEL} - This token indicates the currently reported audio level from the speech engine. The replaced value will be from “0” to “100”.

{STATE_AUDIOLASTFILE} - This token will render the path of the last audio file that is played.

{STATE_AUDIOCOUNT} - This returns the number of all currently-playing audio files. If legacy audio mode is on this value will always be, “0”.

{STATE_AUDIOCOUNT:*variableName / value*} - This returns the number of currently-playing instances of an audio file with a given file path. The parameter can be a text variable name (without quotes: {STATE_AUDIOCOUNT:mySoundVariable}, or it can be a token or a literal if contained within double quotes:
{STATE_AUDIOCOUNT:"{TXT:someTextVariable}"} or
{STATE_AUDIOCOUNT:"C:\Sounds\Robot.wav"}.

Notes: Since sounds run asynchronously in VoiceAttack, there is a slight chance that if you use this token *IMMEDIATELY* after executing a ‘Play a Sound’ action the file may not yet have had a chance to queue or load up and will not be included in the count. This is technically correct, but may not be a proper count depending on what you are trying to accomplish.

This token will allow you to put a literal in that contains colons (for file paths). Use with caution (or use variables/tokens).

If legacy audio mode is on this value will always be, “0”.

{STATE_AUDIOPOS:*variableName / value*} - This returns the position of currently-playing audio file with a given file path, expressed as seconds. The parameter can be a text variable name (without quotes: {STATE_AUDIOPOS:mySoundVariable}, or it can be a token or a literal if contained within double quotes: {STATE_AUDIOPOS:"{TXT:someText}"} or {STATE_AUDIOPOS:"C:\Sounds\TargetEwoks.wav"}.

Notes: Since sounds run asynchronously in VoiceAttack, there is a slight chance that if you use this token *IMMEDIATELY* after executing a 'Play a Sound' action the file may not yet have had a chance to queue or load up and will return a position of "0". This is technically correct, but may not be a proper value depending on what you are trying to accomplish.

Since you can run multiple instances of a sound file at once, if there is more than one instance currently playing, the rendered value will be, "0" (assumptions cannot be made on which instance to be chosen). To help with this, check the {STATE_AUDIOCOUNT:variable} token outlined above prior to using the {STATE_AUDIOPOS} token.

If no instances of the indicated file are currently playing, "0" will be returned.

This token will allow you to put a literal in that contains colons (for file paths). Use with caution (or use variables/tokens).

If legacy audio mode is on this value will always be, "0".

{STATE_AUDIOOUTPUTTYPE} - This renders the currently-selected audio output type as indicated on the Audio tab of the Options screen. The possible rendered values are, "**Legacy**" when, "Legacy Audio" is selected, "**Windows**" when "Windows Media Components" are selected and, "**Integrated**" when, "Integrated Components" is selected.

{STATE_TTSCOUNT} - This renders the number of all currently-playing text to speech synths. If an error is encountered, this value will be rendered as "-1".

{STATE_DEFAULTPLAYBACK} - This returns the device name of the default multimedia audio playback device as indicated by Windows. Note that there is a very minor memory leak when accessing the multimedia device property store, so this will be reflected in VoiceAttack (using this token sparingly will not present a problem… running it over and over in a loop will chew up memory… the search for a better way continues).

{STATE_PLAYBACKDEVICECOUNT} - This renders the number of active playback devices as reported by Windows. If an error is encountered, this value will be rendered as "-1".

{STATE_PLAYBACKDEVICE:*value*} - This renders the playback device name as it relates to its ordinal device number. {STATE_PLAYBACKDEVICE:0} will render the device name of device zero. If an error is encountered, this value will be rendered as empty.

{STATE_DEFAULTPLAYBACKCOMMS} - This works just like {STATE_DEFAULTPLAYBACK}, except the default communications playblack device name will be rendered.

{STATE_DEFAULTRECORDING} - This works just like {STATE_DEFAULTPLAYBACK}, except the default multimedia recording device name will be rendered.

{STATE_RECORDINGDEVICECOUNT} - This renders the number of active recording devices as reported by Windows. If an error is encountered, this value will be rendered as

“-1”.

{STATE_RECORDINGDEVICE:*value*} - This renders the recording device name as it relates to its ordinal device number. {STATE_RECORDINGDEVICE:0} will render the device name of device zero. If an error is encountered, this value will be rendered as empty.

{STATE_DEFAULTRECORDINGCOMMS} - This works just like {STATE_DEFAULTPLAYBACK}, except the default communications recording device name will be rendered.

{STATE_SYSDIR} - This renders the path of the system directory (e.g. ‘C:\Windows\System32’).

{STATE_WINDIR} - This renders the path of the Windows directory (e.g. ‘C:\Windows’).

{STATE_ENV:*textValue*} - This renders the Windows environment variable specified in, ‘textValue’. For example, ‘{STATE_ENV:programfiles}’ would (usually) render ‘C:\Program Files’.

{STATE_CULTURE} - This renders the default user locale of the system.

{STATE_UICULTURE} - This renders the default user interface language.

{STATE_SPEECHCULTURE} - This renders the culture name of the current speech engine that VoiceAttack is using. This value will be empty if speech services are disabled.

{STATE_SYSVOL} - This returns the system volume (default playback device) rendered as a value from “0” to “100”.

{STATE_SYSMUTE} - If the system volume (default playback device) is muted, this token is rendered as “1”. If not, the value is rendered as “0”.

{STATE_MICVOL} - This returns the microphone volume (default recording device) rendered as a value from “0” to “100”.

{STATE_MICMUTE} - If the microphone volume (default recording device) is muted, this token is rendered as “1”. If not, the value is rendered as “0”.

{STATE_APPVOL:*variableName / value*} - This renders a value between “0” and “100” based on the indicated app volume as it relates to the **System Volume Mixer**. If the volume cannot be accessed (app closed or no audio is playing for instance), a value of "-1" will be returned. The parameter for this token should be the window title, process name or window class name of the target application (see “Set Audio Level” section of the, “Other Stuff” screen for more information on targeting the application). The parameter can be a text variable name (without quotes: {STATE_APPVOL:myTextVariable}, or it can be a token or a literal if contained within double quotes: {STATE_APPVOL:"*windows media*"} or {STATE_APPVOL:"{TXT:myTextVariable}"}.

{STATE_APPMUTE:*variableName / value*} - This works exactly the same as the {STATE_APPVOL} token above, except this renders a value of “1” if the indicated application is muted as it relates to the System Volume Mixer, and “0” if the application is not muted. “-1” is rendered if the information cannot be accessed.

{STATE_SPEECHDEVICEMUTE} - This token tests if the recording device that the speech engine is currently using is muted. If the device is muted, “1” will be rendered. Otherwise, the rendered value will be “0”.

{STATE_SPEECHDEVICEVOL} - This returns the volume of the recording device that the speech engine is currently using rendered as a value from “0” to “100”.

{STATE_SPEECHACTIVE} - This token tests to see if the speech engine is detecting speech. If speech is currently detected, the replaced value will be "1". If, not, the replaced value will be "0".

{STATE_VA_VERSION} - Displays the full VoiceAttack version: “1.5.12.15”.
{STATE_VA_VERSION_MAJOR} - Displays the version major component: “1”.
{STATE_VA_VERSION_MINOR} - Minor component: “5”.
{STATE_VA_VERSION_BUILD} - Build component: “12”.
{STATE_VA_VERSION_REVISION} - Revision component: “15”.
{STATE_VA_VERSION_ISRELEASE} - Returns “1” if the version is a release version. “0” if it is not (beta release).
{STATE_VA_VERSION_COMPARE:*textVariable*} - Returns “1” if the VoiceAttack version number is greater than or equal to the version number indicated in the text variable.
“0” if the indicated version is earlier than the VoiceAttack version number.

{STATE_VA_PLUGINSENABLED} - Returns “1” if plugin support is enabled. “0” if not.

{STATE_VA_NESTEDTOKENSENABLED} - Returns “1” if the nested tokens option is enabled. “0” if not.

{STATE_VA_IS64BIT} - Returns “1” if the VoiceAttack process is 64-bit. “0” if not.

Joystick State Token Reference
This section is for the joystick state tokens. Everything below the buttons and POV are lifted directly from DirectX states. Each state value may or may not be available for your device, so, your mileage may vary.

{STATE_JOYSTICKBUTTONS} - This token has been replaced with {STATE_JOYSTICKSHORTCUTS} (see above).

{STATE_JOYSTICK1ENABLED} {STATE_JOYSTICK2ENABLED} {STATE_JOYSTICK3ENABLED} {STATE_JOYSTICK4ENABLED} - these four tokens test whether or not joystick 1, 2, 3 or 4 are enabled in VoiceAttack. If the joystick is enabled, the rendered value will be “1”. If the indicated joystick is not enabled, the value will be “0”.

{STATE_JOYSTICK1ISGAMEPAD} {STATE_JOYSTICK2ISGAMEPAD} {STATE_JOYSTICK3ISGAMEPAD} {STATE_JOYSTICK4ISGAMEPAD} - these four tokens return “1” if the corresponding joystick is designated as a, ‘Gamepad Controller’ when the joystick is assigned in VoiceAttack. If the joystick has not been designated as a ‘Gamepad Controller’, the value rendered will be “0”.

{STATE_JOYSTICK1BUTTON:*buttonNumber*}
{STATE_JOYSTICK2BUTTON:*buttonNumber*}
{STATE_JOYSTICK3BUTTON:*buttonNumber*}
{STATE_JOYSTICK4BUTTON:*buttonNumber*} - these four tokens test to see if particular joystick's button is down or not. The, 'buttonNumber' parameter is the number of the button on the desired stick. To test if button 10 is down on joystick 2, just use this token: {STATE_JOYSTICK2BUTTON:10}. Once again, if the button is down on the tested stick, the replaced value will be "1". If the button is not down, the value will be "0".

{STATE_JOYSTICK1ANYBUTTON} {STATE_JOYSTICK2ANYBUTTON} {STATE_JOYSTICK3ANYBUTTON} {STATE_JOYSTICK4ANYBUTTON} - these four tokens test whether or not any button is pressed on joystick 1,2, 3 or 4. If any button is down, the rendered value will be “1”. If no buttons are pressed down, the value will be “0”.

Point of View (Hat) Controllers – (Note: there are up to 4 available POV controllers per stick)

{STATE_JOYSTICK1POVENABLED} {STATE_JOYSTICK2POVENABLED} {STATE_JOYSTICK3POVENABLED} {STATE_JOYSTICK4POVENABLED} - this token indicates whether or not POV is enabled for the indicated stick. If enabled the replaced value will be “1”. Otherwise, the replaced value will be “0”.

{STATE_JOYSTICK*X*POV*Y*TYPE} - Use this token to find out how the POV is being used by VoiceAttack (as indicated in the joystick options on the options page). X indicates the stick number (1-4) and Y indicates the POV controller (1-4). So, to get the POV type for the second POV controller on stick 1, use: {STATE_JOYSTICK1POV2TYPE}. The replaced value will be one of the following:
“-1” - POV not available.

“0” - POV available, not turned on in settings.
“1” - POV acts as on/off switch (any direction will cause the POV to indicate as switched.
“2” - POV acts as two-directional switch (up or down).
“3” - POV acts as two-directional switch (left or right).
“4” - POV acts as four-directional switch (up, down, left, right).
“8” - POV acts as eight-directional switch (up, up right, right, down right, down, down left, left, up left).

{STATE_JOYSTICKXPOVY} - This token is used to get the direction pressed by the POV controller. X indicates the stick (1-4). Y indicates the POV controller (1-4). So, to get the position value for stick 1, POV controller 2, use: {STATE_JOYSTICK1POV2}.
The replaced value will be “CENTER”, “UP”, “UPRIGHT”, “RIGHT”, “DOWNRIGHT”, “DOWN”, “DOWNLEFT”, “LEFT”, “UPLEFT” depending on the direction pushed, or, “-1” if the POV controller is unavailable. Note that certain directions will only be available due to the type of POV setup in options (see the token above for getting the POV type). For POV type “1” (POV is on/off switch), the only position that will be indicated and that you should test for is, “UP” (this is for speed reasons). For “2” (POV is up/down only), “UP” and “DOWN” are indicated. For “3” (POV is left/right only), “LEFT” and “RIGHT” are indicated. For “4” (four-direction), “LEFT”, “RIGHT”, “UP” and “DOWN” are indicated. For “8” (eight-direction), all eight directions are available.

DirectX Joystick States (simple DirectX query, not tracked by VoiceAttack for managing events):

{STATE_JOYSTICKXPOVY_NUMERIC} - This token is used to get the numeric value presented by the POV controller. X indicates the stick (1-4). Y indicates the POV controller (1-4). So, to get the position value for stick 1, POV controller 2, use: {STATE_JOYSTICK1POV2}. The value will usually be “-1” (some drivers may report “65535”) if the POV is centered, or “0” to “35999” as the POV is pressed in a direction. You can use this token if the VoiceAttack, 'switch' model does not fit your needs.

{STATE_JOYSTICK1X} {STATE_JOYSTICK2X} {STATE_JOYSTICK3X} {STATE_JOYSTICK4X} - This token is used to indicate the X value of the indicated joystick. The replaced value is “0” at minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1Y} {STATE_JOYSTICK2Y} {STATE_JOYSTICK3Y} {STATE_JOYSTICK4Y}- This token is used to indicate the Y value of the indicated joystick. The replaced value is “0” at minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1Z} {STATE_JOYSTICK2Z} {STATE_JOYSTICK3Z} {STATE_JOYSTICK4Z} - This token is used to indicate the Z value of the indicated joystick. The replaced value is “0” at minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ROTATIONX} {STATE_JOYSTICK2ROTATIONX} {STATE_JOYSTICK3ROTATIONX} {STATE_JOYSTICK4ROTATIONX}- This token is used to indicate the rotation X value of the indicated joystick. The replaced value is “0” at

minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ROTATIONY} {STATE_JOYSTICK2ROTATIONY} {STATE_JOYSTICK3ROTATIONY} {STATE_JOYSTICK4ROTATIONY} - This token is used to indicate the rotation Y value of the indicated joystick. The replaced value is “0” at minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ROTATIONZ} {STATE_JOYSTICK2ROTATIONZ} {STATE_JOYSTICK3ROTATIONZ} {STATE_JOYSTICK4ROTATIONZ} - This token is used to indicate the rotation Z value of the indicated joystick. The replaced value is “0” at minimum and “65535” at maximum. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ACCELERATIONX} {STATE_JOYSTICK2ACCELERATIONX} {STATE_JOYSTICK3ACCELERATIONX} {STATE_JOYSTICK4ACCELERATIONX} - This token is used to indicate the acceleration X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ACCELERATIONY} {STATE_JOYSTICK2ACCELERATIONY} {STATE_JOYSTICK4ACCELERATIONY} {STATE_JOYSTICK4ACCELERATIONY} - This token is used to indicate the acceleration Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ACCELERATIONZ} {STATE_JOYSTICK2ACCELERATIONZ} {STATE_JOYSTICK3ACCELERATIONZ} {STATE_JOYSTICK4ACCELERATIONZ} - This token is used to indicate the acceleration Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARACCELERATIONX}
{STATE_JOYSTICK2ANGULARACCELERATIONX}
{STATE_JOYSTICK3ANGULARACCELERATIONX}
{STATE_JOYSTICK4ANGULARACCELERATIONX} - This token is used to indicate the angular acceleration X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARACCELERATIONY}
{STATE_JOYSTICK2ANGULARACCELERATIONY}
{STATE_JOYSTICK3ANGULARACCELERATIONY}
{STATE_JOYSTICK4ANGULARACCELERATIONY} - This token is used to indicate the angular acceleration Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARACCELERATIONZ}
{STATE_JOYSTICK2ANGULARACCELERATIONZ}
{STATE_JOYSTICK3ANGULARACCELERATIONZ}
{STATE_JOYSTICK4ANGULARACCELERATIONZ} - This token is used to indicate the angular acceleration Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARVELOCITYX}
{STATE_JOYSTICK2ANGULARVELOCITYX}
{STATE_JOYSTICK3ANGULARVELOCITYX}
{STATE_JOYSTICK4ANGULARVELOCITYX} - This token is used to indicate the angular velocity X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARVELOCITYY}
{STATE_JOYSTICK2ANGULARVELOCITYY}
{STATE_JOYSTICK3ANGULARVELOCITYY}
{STATE_JOYSTICK4ANGULARVELOCITYY} - This token is used to indicate the angular velocity Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1ANGULARVELOCITYZ}
{STATE_JOYSTICK2ANGULARVELOCITYZ}
{STATE_JOYSTICK3ANGULARVELOCITYZ}
{STATE_JOYSTICK4ANGULARVELOCITYZ} - This token is used to indicate the angular velocity Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1FORCEX} {STATE_JOYSTICK2FORCEX}
{STATE_JOYSTICK3FORCEX} {STATE_JOYSTICK4FORCEX} - This token is used to indicate the force X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1FORCEY} {STATE_JOYSTICK2FORCEY}
{STATE_JOYSTICK3FORCEY} {STATE_JOYSTICK4FORCEY} - This token is used to indicate the force Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1FORCEZ} {STATE_JOYSTICK2FORCEZ}
{STATE_JOYSTICK3FORCEZ} {STATE_JOYSTICK4FORCEZ} - This token is used to indicate the force Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1TORQUEX} {STATE_JOYSTICK2TORQUEX}
{STATE_JOYSTICK3TORQUEX} {STATE_JOYSTICK4TORQUEX} - This token is used to indicate the torque X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1TORQUEY} {STATE_JOYSTICK2TORQUEY}
{STATE_JOYSTICK3TORQUEY} {STATE_JOYSTICK4TORQUEY} - This token is used to indicate the torque Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1TORQUEZ} {STATE_JOYSTICK2TORQUEZ}
{STATE_JOYSTICK3TORQUEZ} {STATE_JOYSTICK4TORQUEZ} - This token is used to indicate the torque Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1VELOCITYX} {STATE_JOYSTICK2VELOCITYX}

{STATE_JOYSTICK3VELOCITYX} {STATE_JOYSTICK3VELOCITYX} - This token is used to indicate the velocity X value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1VELOCITYY} {STATE_JOYSTICK2VELOCITYY} {STATE_JOYSTICK3VELOCITYY} {STATE_JOYSTICK4VELOCITYY} - This token is used to indicate the velocity Y value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICK1VELOCITYZ} {STATE_JOYSTICK2VELOCITYZ} {STATE_JOYSTICK3VELOCITYZ} {STATE_JOYSTICK4VELOCITYZ} - This token is used to indicate the velocity Z value of the indicated joystick. This value will be “-1” if the stick is unavailable.

{STATE_JOYSTICKXSLIDERY} - This token is used to indicate the slider value. X indicates the stick number (1-4) and Y indicates the control number (1 or 2). If the control is not available, the replaced value will be “-1”.

{STATE_JOYSTICKXACCELERATIONSLIDERY} - This token is used to indicate the acceleration slider value. X indicates the stick number (1-4) and Y indicates the control number (1 or 2). If the control is not available, the replaced value will be “-1”.

{STATE_JOYSTICKXFORCESLIDERY} - This token is used to indicate the force slider value. X indicates the stick number (1-4) and Y indicates the control number (1 or 2). If the control is not available, the replaced value will be “-1”.

{STATE_JOYSTICKXVELOCITYSLIDERY} - This token is used to indicate the velocity slider value. X indicates the stick number (1-4) and Y indicates the control number (1 or 2). If the control is not available, the replaced value will be “-1”.

Gamepad Controller Only

{STATE_JOYSTICK1LEFTTRIGGER} {STATE_JOYSTICK2LEFTTRIGGER} {STATE_JOYSTICK3LEFTTRIGGER} {STATE_JOYSTICK4LEFTTRIGGER}
For devices listed as, ‘Gamepad Controller (1-4)’, you can get the left trigger position value from “0” to “255”. This value is “-1” if the stick is not available or if the stick is NOT a gamepad controller.

{STATE_JOYSTICK1RIGHTTRIGGER} {STATE_JOYSTICK2RIGHTTRIGGER} {STATE_JOYSTICK3RIGHTTRIGGER} {STATE_JOYSTICK4RIGHTTRIGGER}
For devices listed as, ‘Gamepad Controller (1-4)’, you can get the right trigger position value from “0” to “255”. This value is “-1” if the stick is not available or if the stick is NOT a gamepad controller.

VoiceAttack Path Tokens

VoiceAttack has a few tokens that can be used in places that require a file path (sound file locations and application locations). There are certain cases where it is helpful to keep certain files together, especially when it comes to sharing profiles.

{VA_DIR} - This is the VoiceAttack installation directory.

{VA_ASSEMBLIES} - This is the VoiceAttack Shared\Assemblies folder that is (should be) located in the VoiceAttack installation folder. Note that no check is made to see if the folder exists.

{VA_SOUNDS} - By default, this is the folder named, 'Sounds' under the VoiceAttack root directory. This is a place where you can store your VoiceAttack sound packs. If you do not want to use the 'Sounds' folder in the VoiceAttack installation directory, you can change this to be whatever folder you want it to be on the VoiceAttack Options page. Note that no check is made to see if this folder exists.

{VA_APPS} - This is the folder named, 'Apps' under the VoiceAttack root directory. This is the place where you can store apps that you use with VoiceAttack (.exe files) and VoiceAttack plugins (.dll files). Just like the, 'Sounds' folder, you can change the location of the VoiceAttack Apps folder in the VoiceAttack Options page. Note that no check is made to see if this folder exists.

Quick Input, Variable Keypress and Hotkey Key Indicators

As explained earlier in this document, you can include special indicators in your Quick Input, keypress variable text and variable hotkey text to represent keys that do not have a character representation (such as, 'Enter', 'Tab', 'F1', etc.).
Below is a list of the acceptable key indicators, some with a brief description. Remember that **key indicators need to be enclosed in square brackets: []**. At the bottom of this list is the Quick Input function list (currently only contains, 'PAUSE', but will grow as needed).

For information on what these are for, see the section titled, 'Quick Input' in the, 'Other Stuff' screen documentation, or the section detailing variable keypresses in the, 'Keypress' screen documentation.

Note: Items below that are marked with a blue asterisk (*) are for Quick Input only. Items that are marked with a red asterisk (*) are for keypress variables only and are not available for Quick Input.

ENTER - presses the enter key
TAB - presses the tab key
ESC - presses the escape key
ESCAPE - works the same as, 'esc' above
BACK - press the backspace button
BACKSPACE - works the same as, 'back' above
SPACE - presses the space bar (note that you can also just have a space in your keypress variable or Quick Input value.

SHIFTDOWN* - holds the left shift key down
SHIFTUP* - releases the left shift key
RSHIFTDOWN* - right shift if you need it
RSHIFTUP*
LSHIFTDOWN* - works the same as shiftdown
LSHIFTUP* - works the same as shiftup

SHIFT* - press left shift key
LSHIFT* - press left shift key
RSHIFT* - press right shift key

ALTDOWN* - holds down the left alt key
ALTUP* - releases the left alt key
RALTDOWN* - holds down the right alt key
RALTUP* - releases the right alt key
LALTDOWN* - works the same as altdown
LALTUP* - works the same as altup

ALT* - keypress variable - press left alt key
LALT* - keypress variable - press left alt key
RALT* - keypress variable - press right alt key

CTRLDOWN* - holds down the left ctrl key

CTRLUP* - releases the left ctrl key
RCTRLDOWN* - holds down right ctrl key
RCTRLUP* - releases right ctrl key
LCTRLDOWN* - works the same as ctrldown
LCTRLUP* - works the same as ctrlup

CTRL* - keypress variable only - press left ctrl key
LCTRL* - keypress variable only - press left ctrl key
RCTRL* - keypress variable only - press right ctrl key

WINDOWN* - holds down the left win key
WINUP* - releases the left win key
RWINDOWN* - holds down the right win key
RWINUP* - releases the right win key
LWINDOWN* - works the same as windown
LWINUP* - works the same as winup

WIN* - keypress variable only - press left win key
LWIN* - keypress variable only - press left win key
RWIN* - keypress variable only - press right win key

DEAD - The, 'dead' key that is available on German keyboards (located next to the '1' key) and various French keyboards (located next to the 'P' key).

NUM0 - numeric pad 0-9
NUM1
NUM2
NUM3
NUM4
NUM5
NUM6
NUM7
NUM8
NUM9

NUM* - numeric pad *
NUM+ - numeric pad +
NUM- - numeric pad -
NUM. - numeric pad .
NUM/ - numeric pad /
NUMENTER - numeric pad enter
NUMINSERT - numeric pad insert
NUMHOME - numeric pad home
NUMDELETE - numeric pad delete
NUMPAGEUP - numeric pad page up
NUMPAGEDOWN - numeric pad page down
NUMEND - numeric pad end
NUMRIGHT - numeric pad right arrow
NUMLEFT - numeric pad left arrow

NUMUP - numeric pad up arrow
NUMDOWN - numeric pad down arrow

F1-F24 - press F(unction) keys... F1-F24

ARROWD - arrow down
ARROWL - arrow left
ARROWR - arrow right
ARROWU - arrow up
CAPSLOCK - toggle capslock
DEL - press delete key
DELETE - works the same as del
END - press end key
HOME - press home key
INS - press insert key
INSERT - works the same as ins
NUMLOCK - toggle numlock
PAGEUP - press page up
PAGEDOWN - press page down
~~PAUSE - press the pause/break button~~ (use, 'BREAK' instead)
BREAK - press the pause/break button
PRINTSCREEN - press printscreen button
SCRLOCK - toggle scroll lock
SCROLLLOCK - works the same as scrlock

VOLUMEMUTE - media volume mute key
VOLUMEDOWN - media volume down key
VOLUMEUP - media volume up key
NEXTTRACK - media next track key
PREVTRACK - media previous track key
STOP - media stop key
PLAYPAUSE - media play/pause key

0 – 255 - As an additional helper, if you *happen* to know the virtual key value, you can put it between square brackets and it will be used. For instance, the virtual key value for the, 'A' key is 65, 'B' is 66 and 'C' is 67. If you put [SHIFTDOWN][65][66][67][SHIFTUP] in Quick Input, 'ABC' will be typed out (note the uppercase characters). Note that you don't save any time by using this… it's just used behind the scenes in other ways and is exposed in this way for you to use ;)

Quick Input inline functions

[PAUSE:*seconds*] - Using this indicator will allow you to insert a pause between characters. Simply use the term, 'PAUSE', followed by a colon, then the amount of time in seconds that you would like to pause. For example, A[PAUSE:2.5]B[PAUSE:0.5]C will press, 'A', then pause 2.5 seconds, then press, 'B', pause one half second, then press, 'C'.

Key State Token Parameter Values

Key state token parameters are used with the {STATE_KEYSTATE:*key*} token. The, 'key' parameter can be any key you can type into a token (A, B, C, #, @, etc). For keys that you *can't* type into a token, use the items from the list below. For instance, if you wanted to see if the F10 key is down, simply use the following token: {STATE_KEYSTATE:F10}.
Note: For convenience, if the key parameter is surrounded by square brackets ('[' and ']'), the brackets will be automatically removed before processing.
Note: *Key* can also accept an integer value that corresponds to a virtual key code. For example, {STATE_KEYSTATE:144} corresponds to VK_NUMLOCK.

See the section on tokens elsewhere in this document for more info.

ENTER
TAB
ESC
ESCAPE
BACK
BACKSPACE
SPACE

DEAD - The, 'dead' key that is available on German keyboards (located next to the '1' key) and various French keyboards (located next to the 'P' key).

LCTRL - Left control key
CTRL - Same as left control key
RCTRL - Right control key

LALT - Left ALT key
ALT - Same as left ALT key
RALT - Right ALT key

LSHIFT - Left shift key
SHIFT - Same as left shift key
RSHIFT - Right shift key

LWIN - Left Windows key
WIN - Same as left Windows key
RWIN - Right Windows key

NUM0 - numeric pad 0-9
NUM1
NUM2
NUM3
NUM4
NUM5
NUM6
NUM7
NUM8

NUM9

NUM* - numeric pad *
NUM+ - numeric pad +
NUM- - numeric pad -
NUM. - numeric pad .
NUM/ - numeric pad /
NUMENTER - numeric pad enter

F1 - F1-F24
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
F13
F14
F15
F16
F17
F18
F19
F20
F21
F22
F23
F24

ARROWD - arrow down
ARROWL - arrow left
ARROWR - arrow right
ARROWU - arrow up
CAPSLOCK - caps lock
DEL - delete key
DELETE - works the same as del
END - end key
HOME - home key
INS - insert key
INSERT - works the same as ins
NUMLOCK - numlock
PAGEUP -page up
PAGEDOWN - page down

PAUSE -pause/break button
PRINTSCREEN - printscreen button
SCRLOCK - scroll lock
SCROLLLOCK - works the same as scrlock
VOLUMEMUTE - media volume mute key
VOLUMEDOWN - media volume down key
VOLUMEUP - media volume up key
NEXTTRACK - media next track key
PREVTRACK - media previous track key
STOP - media stop key
PLAYPAUSE - media play/pause key

VoiceAttack Plugins (for the truly mad)

A VoiceAttack plugin is code that resides in a dynamic-link library (.dll) that VoiceAttack can call at any point in a command. The plugin can be code that you (or somebody else) can write to enhance the capabilities of VoiceAttack. A plugin will allow you to pass information from VoiceAttack into your code, execute whatever features you want, then allow your code to interact with VoiceAttack or pass information back to VoiceAttack for further processing within your commands.

This is an experimental part of VoiceAttack and is really not intended for everyone (that's why the documentation is all the way down here near the end). This is a means to be able to make VoiceAttack do pretty much whatever we want it to do without affecting its core functionality. The interface for the plugins will be available to anybody that wants to jump in, and I am hoping that what is provided will be easy to understand. Please note that this is an evolving endeavor, and not every aspect of this feature is (or will be) explained here. More aspects will be uncovered/discovered as (or if) more people decide to use this feature. This documentation may or may not be accurate or up to date, depending on the version of VoiceAttack you are using. This document covers version 4 of the plugin interface. To read about previous versions of this interface, you will need to obtain an earlier version of VoiceAttack or post a request in the forum as somebody may have a copy lying around.

Setup

If you have the right tools, a VoiceAttack plugin should be fairly easy to construct. The only rub is that there is an interface that you must adhere to (more later). Sample code will be placed in a folder called, 'Plugin Samples' in the VoiceAttack installation directory. Note that there is no library for you to reference. Note: Although having a library with interfaces to reference would speed things up slightly, the decision was made to not require you have to tote around additional files to get your plugin to work. As you will see, there's very little you'll have to do or remember to get information back and forth to VA, or to make VA perform some type of action.

Basically, all you will need at this point in time is a version of Visual Studio that supports at least .Net Framework 4.0 (you pick the language). If you do not have this, you can go to Microsoft's site and download an Express/Community edition for **free**. Note that there is a quick primer later on in this section to help you set up your environment if you have not built .dlls before. Just search for '**Notes on testing your plugin / setup**'.

If you are planning on sharing your plugin with others, it is suggested that you compile your .dll using the, 'Any CPU' platform target. Also, you may even want to consider making your plugins open-source, due to trust/security concerns. Some people will not run a compiled .dll unless they know that the source is trusted and/or know what the code looks like (I am one of those people).

Turning on plugin support in VoiceAttack

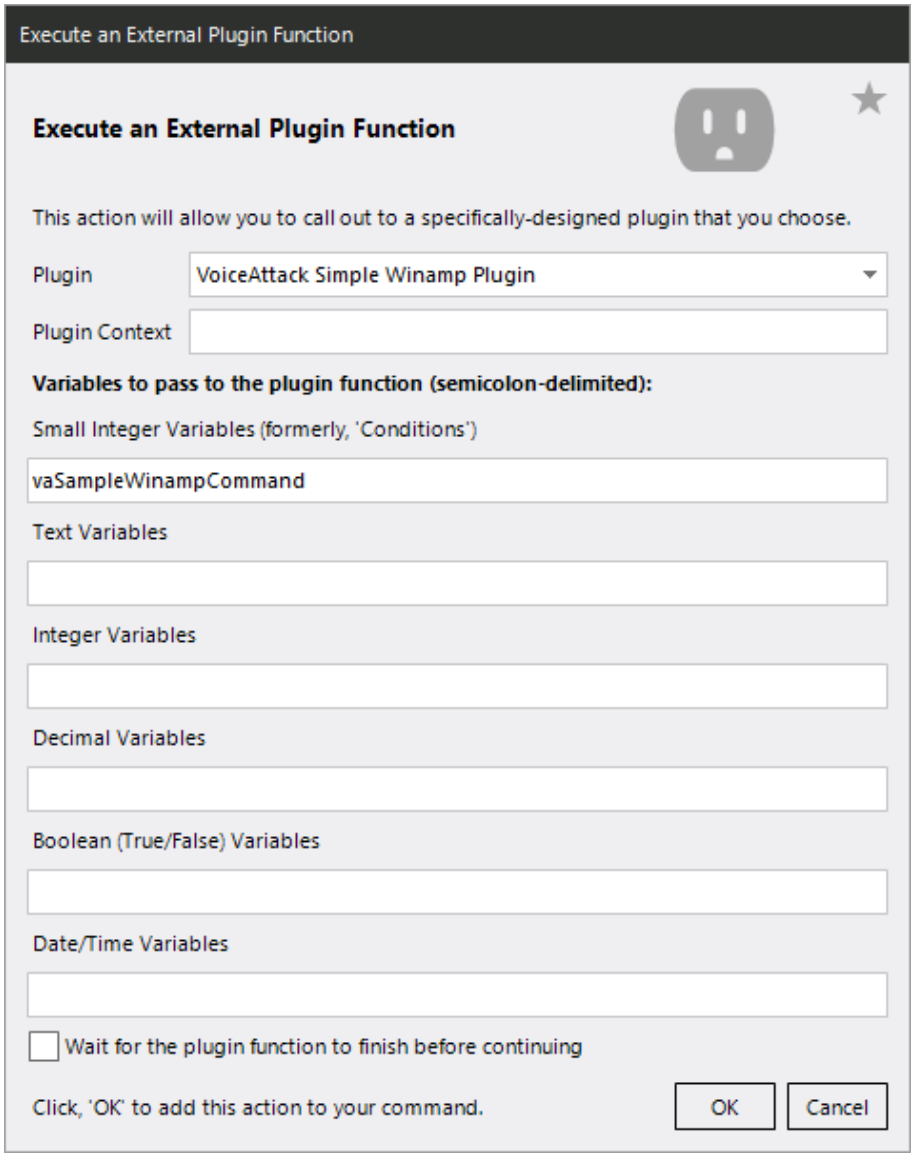
To turn on plugin support in VoiceAttack, you need to go into the options screen and check the, 'Enable Plugin Support' checkbox. A nice warning message will come up and advise you of the dangers of using plugins that somebody else creates. If you accept and continue, you'll get another box that indicates that VoiceAttack will need to be restarted to initialize any plugins. When you run VoiceAttack with plugin support enabled, you will see log messages indicating the state of each plugin that was found and validated.

Turning off plugin support

There are two ways to disable plugin support. You can deselect the box described above, or, you can launch VoiceAttack with the control and shift buttons pressed at the same time. This is a way to catch VoiceAttack before the plugin initializers can be called (in case of danger o_O). You can always just wipe your apps directory if you REALLY want to make sure nothing is going to happen ;)

Running a plugin from within VoiceAttack

To invoke a VoiceAttack plugin, you will need to put an, 'Execute an External Plugin Function' (long name... might change... lol) command action in your command:



On this screen, you will pick the plugin that you want to use and any condition or text values you want passed in. You can also indicate a context value and indicate whether or not you want VoiceAttack to wait for the plugin function to return.

NOTE: Any reference below this point regarding the variable input boxes (Small Integer, Text, Integer, Decimal, Boolean and Date/Time) are there for backward compatibility prior to version 4 of the plugin interface. They are not necessary for use after version 4 (VoiceAttack version 1.6+), as you will be able to directly access variables from within your plugin.

****
When the command action is executed, the plugin context value, any small integer variables (indicated by name in a **semicolon-delimited list**), any text, integer, decimal, Boolean or date/time variables (indicated the same way), and plugin state are passed to the plugin's invoke function (currently, 'VA_Invoke1').

At this point, this is where most of the business will (should) occur. The plugin **context** that is passed to the plugin can be any string value that you want it to be or any combination of tokens (fairly simple). Small integer (formerly, 'Condition') variables that are passed in can be altered within the plugin or you can add more small integer values (setting a small integer value to null will remove the small integer on return). Same goes for the other variable types (text, integer, decimal, Boolean, date/time). Small integer, text, integer, decimal, Boolean and date/time variables are global and available to all plugins, so play nice. The dictionary that contains state values can be modified however you want it. It's private to your plugin (no other plugin has access to it, and only your plugin can modify it). Since your plugin class is using static functions, this is provided to simply maintain state privately if you need it. I'm sure you can come up with all kinds of ways to persist state between calls, but this might make things a little bit easier.

After your code goes out of scope and control returns to VoiceAttack, VoiceAttack will update its copy of your plugin state, update any variables you altered and remove any variables set to null.

If you did not check the, **'wait until function returns'** checkbox, the call to the plugin will be tossed into another thread and VoiceAttack will continue processing actions immediately.

If you do check the, **'wait until function returns'** checkbox, VoiceAttack will do just that... wait until your code finishes before continuing. At this point you have a chance to work with any of the values you had modified/added in subsequent actions within the command. For instance, maybe your plugin goes out to the internet to retrieve information. Your plugin can set that information to a value that is passed back to VoiceAttack and then VoiceAttack can use that value (for TTS or for conditional flow).

VoiceAttack Plugin Requirements and Interface (this will be reorganized properly at some point)

The code that you compile **must be a .net framework 4.0+ .dll.**

The .dll must be in a specific location.
The VoiceAttack Apps directory is located (by default) in the VoiceAttack installation directory (this location can be changed in the Options page). I called it, 'Apps' because it's not just for plugins. It can be for your out-of-process (.exe) stuff as well... just thought you should know that for some reason o_O.

VoiceAttack will crawl all **subdirectories** of the Apps directory. VoiceAttack will ONLY crawl subdirectories of the Apps directory, and not the Apps directory itself (that means that any .dll files that are just sitting in the Apps directory will be skipped... this is for

housekeeping reasons... your plugins need to be in their own directories). Again... **The compiled plugin .dll must reside IN A SUBDIRECTORY of the VoiceAttack Apps directory:**

“C:\Program Files (x86)\VoiceAttack\Apps\myPlugin.dll” will be skipped.
“C:\Program Files (x86)\VoiceAttack\Apps\MyPluginDirectory\myPlugin.dll” will be picked up.

When VoiceAttack discovers a .dll, it interrogates it and sees if it is compatible with the current version of VoiceAttack. If it is not, you will see a log entry indicating this.

Functions your plugin must provide...

VoiceAttack will not be instantiating any objects in your plugins, so **all the required functions must be static**. If you have objects that need to be instantiated, feel free to do that from within the static functions. You can call your classes anything you want. VoiceAttack will find all the classes in your .dll that meet the criteria and reference each class as a separate plugin (this way if you have multiple plugins per assembly (.dll)).

As stated above, your class must contain static functions. For version 4 (the current version), seven of them are currently necessary. The functions are for display name, plugin id, extended display info, initialization, invoke and application exit, and command stop. Each of the functions must have a specific set of parameters to work.

The first required plugin function is called **VA_Id**. This returns a Guid to VoiceAttack and uniquely identifies your plugin so that it can actually be called when needed.
This is a Guid that **must be generated by you**. Please find a proper tool (Visual Studio has one built in) that will generate the value for you. Note that if your project has multiple classes that present themselves as plugins, the Guid must be different for each class.

The next plugin function is called, **VA_DisplayName**. This returns a string value and is what VoiceAttack will display when referring to your plugin. You will see this value in the selection lists and log entries.

The third plugin function is called, **VA_DisplayInfo**. This is another simple string value that can be used to display extra information about your plugin (such as the author or helpful info regarding the plugin). Make sure the string that is returned contains proper formatting. This function can return an empty string.

The fourth function is **VA_Init1 (UPDATED IN VERSION 4)**. This function gets called when VoiceAttack starts, before the main screen form is loaded. This function does not return a value (void), however, it has a single dynamic parameter called, ‘vaProxy’. This parameter takes the place of the eight parameters that were used before version 4. Note that this function is called asynchronously and there is no guarantee of the order that plugins will be initialized. Note that through the vaProxy variable, you can access the state values, any VoiceAttack variables as well as be able to execute commands or change profiles (and lots more… more about vaProxy later).

The fifth function is **VA_Exit1 (UPDATED IN VERSION 4)**. This function gets called when

VoiceAttack closes. This will give you an opportunity to clean up anything that you want to upon application exit. This function does not return a value (void), however, it has a single dynamic parameter called, ‘vaProxy’. This parameter takes the place of the state dictionary that was used before version 4. You can use the vaProxy variable to access the state information that is passed back and forth to VoiceAttack (again, more about vaProxy later). This function is called asynchronously and there is no guarantee of the order that plugins will be called on exit.
Note: The, ‘vaProxy’ that is available in VA_Exit1 is limited to being only able to get state information. Any other use of vaProxy in VA_Exit1 may result in an exception being raised and VoiceAttack becoming hung on exit (possibly requiring you to terminate VoiceAttack via the Task Manager).

The sixth required static function is **VA_StopCommand (NEW IN VERSION 4).** This function takes no parameters and is called any time the user executes a, ‘stop all commands’ action or clicks on the, ‘stop all commands’ button on the main screen. This function’s purpose is to provide a means for the plugin to know that this event as occurred. Again, this will be called asynchronously and there is no guarantee of the order that plugins will be called.

The seventh required function is **VA_Invoke1 (UPDATED IN VERSION 4)**. This function gets called via a special command action in VoiceAttack ('Execute external plugin function'). The VA_Invoke1 function does not have a return value (void). This function has a single dynamic parameter called, ‘vaProxy’ which replaces all of the parameters for this function that existed prior to version 4.

Plugin parameter notes

Note that this section is now in two parts: version 4 and above and version 3 and below.

Version 4 and above plugin parameter notes

vaProxy

Version 4 of the plugin interface brings an attempt to simplify things somewhat by eliminating an ever-growing list of parameters for each function. In version 3 and versions prior, there was a parameter for the context, the state and each and every data type to be passed in (see v3 notes below on how to access vaProxy in v3 and prior). In this version, we have the single dynamic parameter called, 'vaProxy'. It's not as easy to see what is going on with this approach, but it will make things far more flexible going forward. That is, the vaProxy's capabilities can expand indefinitely without having to rework this interface every time there is a change. Below is an outline of the properties and methods available to vaProxy with a description of how to use each one. Also, if the property or method relates to functionality available in previous versions of the interface you will find notes on that as well.

To use the vaProxy variable, simply access its attributes just as you would any other class. It will be a little awkward at first, as the type of vaProxy is, 'dynamic'. That means that the attributes are not available until runtime (also known as, 'late bound') and you will not have the assistance of Intellisense. You will generally not receive compiler errors when using dynamic variables. If something is wrong, you will receive a runtime error when that line is hit. Again, if you are down here reading this, I am most likely telling you stuff you already know. Rock on ;)

vaProxy Attributes

VoiceAttack's proxy object, 'vaProxy', has a bunch of methods and properties. Some of these methods and properties are accessed directly from the proxy object itself: vaProxy.Context, vaProxy.GetText(string VariableName). The proxy object also has a set of objects to help organize its properties and methods into logical groups. These objects are currently Command, Profile, Queue, Utility, Dictation and State. Each of these will be outlined below.

vaProxy Base Attributes

These are the attributes that belong directly to the vaProxy object. They represent the basic functionality of the proxy object that you've been using for like forever. You call them straight from the vaProxy object:

Context - This is a read-only string property which is set via the, 'Plugin Context' input box on the, 'Execute External Plugin Function' screen. This property is only available when you

are using VA_Invoke1 (it is not accessible in any other function).

Since there is only one function (VA_Invoke1) that is accessed by commands, you need a way to differentiate between different types of requests. This is just a simple way to get information to your plugin without having to assign a variable. You will probably use this property the most.

```
public static void VA_Invoke1(dynamic vaProxy)
{
    if (vaProxy.Context == “lights on”)
        //do some stuff here
    else if (vaProxy.Context == “lights off”)
        //do something different
}
```

This property will be available in VA_Invoke1 only and will be null if accessed elsewhere (including inline functions (see, ‘Execute an Inline Function’ section earlier in this document)).

The Context property takes the place of the Context parameter that was available in VA_Invoke1 for version 3 and before.

SessionState – this is a read/write Dictionary of (String, Object). This property is accessible through VA_Init1, VA_Invoke1 and VA_Exit1. This property is for your own private use within the plugin you create. No other plugin has access to the values. It serves as a kind of session, so that you can easily maintain information between calls without having to do any kind of persistence. The dictionary is (String, Object) so you can name your values whatever you want, as well as store any kind of value type. Whatever you do to this dictionary will be reflected in VoiceAttack upon return (VA_Invoke1 and VA_Init1 only). So, if you empty this dictionary, VoiceAttack's copy of this dictionary will be emptied. Null values will **not** be cleaned out. Note: When the state dictionary is initialized for a plugin, there are three key/value pairs that are included for use. The keys are below (the key names are the same as the token values used elsewhere):
VA_DIR : The installation directory of VoiceAttack.
VA_APPS : VoiceAttack apps/plugins directory.
VA_SOUNDS : VoiceAttack sounds directory.

Again, these values can be erased and manipulated however you want.

```
public static void VA_Init1(dynamic vaProxy)
{
    vaProxy.SessionState.Add("new state value", 369);
    vaProxy.SessionState.Add("second new state value", "hello");

    String sValue = vaProxy.SessionState[“new state value”];
}
```

The SessionState property takes the place of the State parameter that was available for each function in version 3 and before.

NOTE: This property should only be used with plugins, as the values are not maintained

between calls using inline functions.

SetSmallInt(string *VariableName*, short? *Value*) – this method allows you to set a VoiceAttack short integer variable indicated by *VariableName* to a value specified by *Value*. Short integers were referred to as, 'Conditions' in earlier versions of VoiceAttack. Note that the Value can be **null** which will clear out the named variable.

```
public static void VA_Invoke1(dynamic vaProxy)
{
    vaProxy.SetSmallInt("mySmallInt", 55);
}
```

Small integers are public and can be accessed from any plugin or any command within the VoiceAttack user interface. That means that anybody can view or modify these values. You can access the values in conditions using the {SMALL:*variableName*} ({COND:*conditionName*} remains for backward compatibility) token in various places in VoiceAttack.

SetSmallInt takes partial place of the SmallIntegerValues parameter (dictionary of (String, short)) in version 3 and before.

GetSmallInt(string *VariableName*) – this function will return a short integer value if the variable indicated by *VariableName* exists, or **null** if the variable does not exist. Note that you will always want to check to see if the value is null:

```
public static void VA_Invoke1(dynamic vaProxy)
{
    short? myShort = vaProxy.GetSmallInt("mySmallInt");
    if (myShort.HasValue)
    {
        //do some stuff
    }
}
```

GetSmallInt takes partial place of the SmallIntegerValues parameter (dictionary of (String, short)) in version 3 and before.

The functionality you see in GetSmallInt and SetSmallInt is repeated for the following functions with their corresponding data types. Note that each replaces their corresponding dictionary parameter in version 3 and before:

SetInt(string *VariableName*, int? *Value*) – Set a nullable integer value
GetInt(string *VariableName*) – Returns a nullable integer value

SetText(string *VariableName*, string *Value*) – Set a string value
GetText(string *VariableName*) – Returns a string value

SetDecimal(string *VariableName*, decimal? *Value*) – Set a nullable decimal value
GetDecimal(string *VariableName*) – Returns a nullable decimal value

SetBoolean(string *VariableName*, Boolean? *Value*) – Set a nullable Boolean value
GetBoolean(string *VariableName*) – Returns a nullable Boolean value

SetDate(string *VariableName*, DateTime? *Value*) – Set a nullable DateTime value
GetDate(string *VariableName*) – Returns a nullable DateTime value

ProfileNames() - Returns a string array of all profile names.

ProfileInternalIDs() - Returns a GUID array of all non-null profile internal ids.

WriteToLog(String *Value*, String *Color*) – This is a simple way to get some information to the VoiceAttack log. This could be for your own debugging reasons or it could be information for the user. The text indicated in *Value* will be written, with a status icon of the color you choose. The choices are: “red”, “blue”, “green”, “yellow”, “orange”, “purple”, “blank”, “black”, “gray”, “pink”.

```
vaProxy.WriteToLog(“What is love?”, "red");
```

ClearLog() - This simply clears the VoiceAttack log on the main screen.

SetOpacity(int value) - This function will set the main screen’s opacity level from 0 to 100.

ProxyVersion – This will return a System.Version object to indicate the version of the proxy interface. This will help you determine whether or not the installed interface is compatible with your plugin.

```
System.Version v = vaProxy.ProxyVersion;
```

VAVersion – This will return a System.Version object to indicate the version of VoiceAttack currently in use. This will help you determine whether or not the user can use your plugin.

```
System.Version v = vaProxy.VAVersion;
```

PluginPath() - This will return a string that contains the full path of the executing plugin. This function is not applicable when using the proxy object within inline functions (see, ‘Execute an Inline Function’ section earlier in this document). Note: This is a function and not a property for some reason o_O.

```
String s = vaProxy.PluginPath();
```

Stopped - This will return true if the user has clicked the, ‘stop all commands’ button on the main screen or a, ‘Stop all commands’ action has been issued. This property was created mainly for use within inline functions (see, ‘Execute an Inline Function’ section earlier in this document), but works just as well within plugins. Note - the use of this property this can be replaced with the, ‘CommandsStopped’ event (later in this section).

```
Boolean b = vaProxy.Stopped;
```

InstallDir - This is the VoiceAttack installation directory as a string.

```
String s = vaProxy.InstallDir;
```

SoundsDir - By default, this is the folder named, 'Sounds' under the VoiceAttack root directory. This is a place where you can store your VoiceAttack sound packs. If you do not want to use the 'Sounds' folder in the VoiceAttack installation directory, you can change this to be whatever folder you want it to be on the VoiceAttack Options page. Note that no check is made to see if this folder exists. This property returns a string.

AppsDir - This is the folder named, 'Apps' under the VoiceAttack root directory. This is the place where you can store apps that you use with VoiceAttack (.exe files) and VoiceAttack plugins (.dll files). Just like the, 'Sounds' folder, you can change the location of the VoiceAttack Apps folder in the VoiceAttack Options page. Note that no check is made to see if this folder exists. This property returns a string.

AssembliesDir - This is the VoiceAttack Shared\Assemblies folder that is (should be) located in the VoiceAttack installation folder. Note that no check is made to see if the folder exists. This property returns a string.

ResetStopFlag() - This function will reset the flag used to check if the, ‘Stop all commands’ button has been pressed, or a, ‘Stop all commands’ action has been issued. This function was created mainly for use within inline functions (see, ‘Inline Functions’ section later in this document), but works just as well within plugins. Note - the use of this function this can be replaced with the, ‘CommandsStopped’ event (later in this section).

```
if (vaProxy.Stopped)
{
    vaProxy.WriteToLog(“VoiceAttack commands have stopped!”, “Orange”);
    vaProxy.ResetStopFlag();  //reset the flag here
}
```

IsRelease - This Boolean property will be true if the current version of VoiceAttack is a full release and false if it is not.

IsTrial - This Boolean property will be true if the current version of VoiceAttack is the trial version and false if it is not.

PluginsEnabled - This Boolean property will be true if plugins are enabled on the Options screen, and false if plugins are not enabled (the intended use for this is within inline functions).

NestedTokensEnabled - This Boolean property will be true if the, ‘Use nested tokens’ option is enabled on the Options screen and false if not. This is useful when using the Utility.ParseTokens() function.

AutoProfileSwitchingEnabled - This Boolean property will be true if the, ‘Enable Automatic Profile Switching’ option is enabled on the Options screen and false if not.

MainWindowHandle - This returns VoiceAttack’s main window handle as an IntPtr.

```
IntPtr hWnd = vaProxy.MainWindowHandle;
```

Close() - This function will close VoiceAttack. It is the same as if you had clicked on the main screen's close button (top-right).

vaProxy.Utility Attributes

Below are the attributes that belong to the Utility object of vaProxy. It is a collection of functions that are available in VoiceAttack and are exposed so that you may get use out of them. For example, to access the ParseTokens method of the Utility object, you just call it this way: vaProxy.Utility.ParseTokens("some value").

ExtractPhrases(string *Phrases, bool TrimSpaces (optional), bool Lowercase (optiona)*) - This utility function returns a string array containing the extracted phrases indicated by *Phrases*. *Phrases* can be a string containing single, multipart and dynamic phrases that will be broken up into a string array of single phrases the same way that VoiceAttack extracts them. For example, 'hello;hi;howdy' will result in an array with three elements: hello, hi and howdy. 'good[morning;day;night;gravy]' will result in an array with four elements: good morning, good day, good night, good gravy. The optional parameter, 'TrimSpaces' removes any leading and trailing spaces for each element (default is false). Note: Use of dynamic phrases with this function will result in spaces being trimmed regardless of this parameter.
The optional parameter, 'Lowercase' sets every element to lowercase (default is false).

```
String[] myArray = vaProxy.Utility.ExtractPhrases("'good[morning;day;night;gravy]");
```

ParseTokens(String *Value*) – This is just a shortcut to getting values from any of the available tokens.

```
String s = vaProxy.Utility.ParseTokens("{ACTIVEWINDOWTITLE}");
```

CapturedAudio(int type) - This returns a System.IO.MemoryStream (wave stream) that contains audio captured from VoiceAttack's input. The integer parameter is the type of audio that you want to retrieve:
0 – The last recognized audio (that is, audio that was recognized as a spoken command phrase).
1 – The previous recognized audio.
2 – The latest unrecognized audio.
3 – The latest captured recognized or unrecognized audio.
4 – The previous captured recognized or unrecognized audio.
5 – Dictation audio (dictation audio captured while dictation mode is turned on).

This example shows how to play back the latest recognized audio (note the 0 parameter):

```
using (System.IO.MemoryStream ms = vaProxy.Utility.CapturedAudio(0))
{
    if (ms == null)
        return;
    using (System.Media.SoundPlayer sp = new System.Media.SoundPlayer(ms))
```

```
        {
            sp.Play();
        }
    }
```

Note that you will have to dispose of the memory stream yourself.

See also, ‘Captured Audio’ feature earlier in this document.

ResetSpeechEngine() - This resets the current speech engine (without resetting the entire profile).

```
vaProxy.Utility.ResetSpeechEngine();
```

GetSpeechRecordingDeviceMute() - This Boolean function return true if the device that the speech engine is currently using is muted and false if it is not.

SetSpeechRecordingDeviceMute(Boolean *Mute***)** - This sets the mute state of the recording device that the speech engine is currently using. Passing a true value mutes the device. Passing a false value unmutes the device.

ActiveWindowTitle() - This string function returns the active window's title text.

ActiveWindowProcessName() - This string function returns the active window's process name (the one you see in Task Manager).

ActiveWindowProcessID() - Returns the active window's process id (the one you see in Task Manager details) as an integer.

ActiveWindowPath() - This string function returns the path of the active window's executable.

ActiveWindowWidth() - Returns the active window's width as an integer. Helps with resizing/moving.

ActiveWindowHeight() - Returns the active window's height as an integer. Helps with resizing/moving.

ActiveWindowLeft() - Returns the active window's left (X coordinate) as an integer. Helps with resizing/moving.

ActiveWindowTop() - Returns the active window's top (Y coordinate) as an integer. Helps with resizing/moving.

ActiveWindowRight() - Returns the active window's right (left + width) as an integer. Helps with resizing/moving.

ActiveWindowBottom() - Returns the active window's bottom (top + height) as an integer. Helps with resizing/moving.

ProcessExists(String ProcessName) – This Boolean function returns true if a process with the name specified in textVariable exists. Returns false if not. Note that this can take wildcards (*). For instance, if ProcessName contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

ProcessCount(String ProcessName) – Returns an integer value of 1 or more if processes with the name specified in ProcessName exist. Returns 0 if there are none. Note that this can take wildcards (*). For instance, if ProcessName contains '*notepad*' (without quotes), the search will look for processes that have a name that contains 'notepad'.

ProcessForeground(String ProcessName) – This Boolean function returns true if a process with a main window with the title specified in ProcessName is the foreground window. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if ProcessName contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

ProcessMinimized(String ProcessName) – Returns a Boolean value of true if a process with a main window with the title specified in ProcessName is minimized. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if ProcessName contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

ProcessMaximized(String ProcessName) - Returns a Boolean value of true if a process with a main window with the title specified in ProcessName is maximized. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if ProcessName contains '*notepad*' (without quotes), the search will look for a process that has a name that contains 'notepad'.

WindowExists(String WindowName) - Returns a Boolean value of true if a window with the title specified in WindowName exists. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if WindowName contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

WindowForeground(String WindowName) - Returns a Boolean value of true if a window with the title specified in WindowName is the foreground window. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if WindowName contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

WindowMinimized(String WindowName) - Returns a Boolean value of true if a window with the title specified in WindowName is minimized. Returns false if not. Note that this can take wildcards (*) for window titles that change. For instance, if textVariable contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

WindowMaximized(String WindowName) - Returns a Boolean value of true if a window with the title specified in WindowName is maximized. Returns false if not. Note that this

can take wildcards (*) for window titles that change. For instance, if WindowName contains '*notepad*' (without quotes), the search will look for a window that has a title that contains 'notepad'.

WindowCount(String WindowName) - Returns an integer value of 1 or more if windows with the title specified in WindowName exist. Returns 0 if there are none. Note that this can take wildcards (*) for window titles that change. For instance, if WindowName contains '*notepad*' (without quotes), the search will look for windows that have a title that contains 'notepad'.

WindowTitleUnderMouse() - Returns the title for the window that is currently located under the mouse as a string.

WindowProcessUnderMouse() - Returns the process name for the window that is currently located under the mouse as a string.

CommandTargetForeground() - Returns a Boolean value of true if the current command's target is the foreground window. Returns false if the target is not the foreground window or if the target does not exist.

CommandTargetMinimized() - Returns a Boolean value of true if the current command's target is minimized. Returns false if the target is not minimized or if the target does not exist.

CommandTargetMaximized() - Returns a Boolean value of true if the current command's target is maximized. Returns false if the target is not maximized or if the target does not exist.

MousePositionScreenX() - Returns the X coordinate of the mouse position as it relates to the screen as an integer. Zero would be the top-left corner of the screen.

MousePositionScreenY() - Returns the Y coordinate of the mouse position as it relates to the screen as an integer. Zero would be the top-left corner of the screen.

MousePositionWindowX() - Returns the X coordinate of the mouse position as it relates to the active window as an integer. Zero would be the top-left corner of the window.

MousePositionWindowY() - Returns the Y coordinate of the mouse position as it relates to the active window as an integer. Zero would be the top-left corner of the window.

CapsLockOn() - Returns a Boolean value of true if the caps lock key is locked.

NumLockOn() - Returns a Boolean value of true if the numlock key is locked.

ScrollLockOn() - Returns a Boolean value of true if the scroll lock key is locked.

MinimizeUI() - Minimizes the VoiceAttack user interface.

RestoreUI() – Restores a minimized VoiceAttack user interface.

vaProxy.Command Attributes

Below are the attributes that belong to the Command object of vaProxy. These attributes pertain to commands (including the command that has invoked the plugin function or inline function).

Name() – returns a string that indicates the command that is running. If the command was executed using a spoken phrase, what was spoken will be returned. If the command is executed using any other means (keyboard, mouse, joystick, etc.) the full command name will be returned (works exactly like the '{CMD}' token).

```
string commandName = vaProxy.Command.Name();
```

InternalID() – returns a nullable Guid (Guid?) that indicates the internal id specified by the profile author.

```
Guid? commandId = vaProxy.Command.InternalID();
if (commandId.HasValue) //if the internal id was indicated
{
    //do something with the ID here
}
```

Segment(int *iSegment*) – If your command contains, 'dynamic command sections' (see, 'Dynamic Command Sections' in the, 'Command Screen' documentation above), you can retrieve specific portions of the spoken command, indicated by its numeric position. This will allow you to make more precise decisions based on what was spoken.

For instance, let's say you have a complex dynamic command that you use to build several kinds of items like this: 'build [1..10][bulldogs;wolves;strikers][please;]' Then, you say, 'build 5 bulldogs' to execute that command. To find out what was built, you can check Segment(2) (note that the specified, 'segment' is zero-based. So, the first segment is 0. The second is 1, and so on). The returned value will be, 'bulldogs'. Then, you can find out how many items to build by checking Segment(1). The returned value will be, '5' (which you can convert and use in a loop, for instance). For bonus points, you can check Segment(3) and see if, 'please' was spoken and then thank the user for being so polite (or chide them if they didn't say, 'please') ;)

Action() – returns a string to indicate the method by which the current command was executed. The possible results are, 'Spoken', 'Keyboard', 'Joystick', 'Mouse', 'Profile', 'External', 'Unrecognized', 'ProfileUnloadChange', 'ProfileUnloadClose', 'DictationRecognized', 'Plugin' and 'Other'. The value will be, 'Spoken' if the command was executed by a spoken phrase, 'Keyboard' if the command was executed using a keyboard shortcut, 'Joystick' if executed by a joystick button, 'Mouse' if executed by a mouse button click and 'Profile' if the command was executed on profile load (from the command indicated in the profile options screen). The value will be 'External' if the command is executed from a command line or if you right-click on a command on the profile screen and execute it from there. 'Unrecognized' will be the value if the command was executed using the unrecognized phrase catch-all command in the profile options screen. 'ProfileUnloadChange' and 'ProfileUnloadClose' will be returned if the command is executed as part of the profile being unloaded, either by changing the profile or by

VoiceAttack shutting down. ‘DictationRecognized’ is returned if the command was invoked as a result of a dictation phrase being recognized. A value of, ‘Plugin’ is returned if the command is executed from a plugin or inline function. 'Other' is reserved.

Before() – returns a string. When using wildcards in spoken phrases, this is the text that occurs before the wildcard phrase. For example, if using, '*rocket*' as your wildcard phrase and you say, 'i am going for a ride in my rocket ship' Before() will contain, 'i am going for a ride in my' and After() will contain 'ship'.

After() – returns a string. When using wildcards in spoken phrases, this is the text that occurs after the wildcard phrase.

WildcardKey() – returns a string. When using wildcards in spoken phrases, this is the text that is not variable. Using the, ‘*rocket*’ example above, the value returned by this function will be, ‘rocket’.

Confidence() – returns an int that indicates the confidence level that the speech engine provides when the speech engine detects speech. The value range for a spoken command is from 0 to 100. This value will always be 0 when a command is not executed as a spoken command (use the, ‘Action()’ function to check how the command was executed). Note this value is accessible from within the specified unrecognized catch-all command.

MinConfidence() – returns an int that indicates the minimum confidence level set by the user as it applies to the executing command. The value range is 0 to 100. This value will be 0 if the minimum confidence level is not set. Note that this function can return a value even if the command is not spoken.

IsSubcommand() – a Boolean that will be true if the currently-executing command is executing as a subcommand (a subcommand is a command that is executed by another command). The returned value will be false if the command is not a subcommand.

IsDoubleTapInvoked() – a Boolean that will be true if the currently-executing command is executing as a result of a double tap. The returned value will be false if the command is not executed as a result of a double tap.

IsLongPressInvoked() – a Boolean that will be true if the currently-executing command is executing as a result of a long press. The returned value will be false if the command is not executed as a result of a long press.

WhenISay() – returns a string that is the full value of what is indicated in the, ‘When I Say’ input box on the command screen.

IsListeningOverride() – returns a Boolean value of true if the executing command was invoked by a listening override keyword (for instance, if you executed the command by saying, 'Computer, Open Door' instead of, 'Open Door'). Otherwise, the result will be false.

IsComposite() – returns a Boolean value of true if the executing command is composite (where there is a prefix and a suffix). Otherwise, it will be false.

PrefixPart() – returns a string. If an executing command is composite (where there is a prefix and a suffix), this function will return the prefix portion of the command. If called within a non-composite command, this will result in a blank string.

SuffixPart() – returns a string. If an executing command is composite (where there is a prefix and a suffix), this function will return the suffix portion of the command. If called within a non-composite command, this will result in a blank string.

CompositeGroup() – returns a string. If an executing command is composite (where there is a prefix and a suffix), this function will return the group value if it is being used.

Category() – returns a string that indicates the category of the executing command. If this token is used on a composite command (a command using a prefix and suffix), the result will be the category of the prefix and the category of the suffix separated by a space. For the individual parts of a composite category, see, 'PrefixCategory()' and 'SuffixCategory()' below.

PrefixCategory() – returns a string that indicates the prefix category of the executing command (if the executing command is a composite command).

SuffixCategory() – returns a string that indicates the suffix category of the executing command (if the executing command is a composite command).

SetSessionEnabled(string *CommandName, Boolean Enabled*) - This function temporarily sets the enabled state of a command (indicated by name) during the current session (that is, while VoiceAttack is running – the setting is not saved). If a command is able to be executed (by voice, keyboard shortcut, mouse button, etc.), setting *Enabled* to false will temporarily disable the command from executing. Setting *Enabled* to true again will re-enable the command. Note: You will still be able to execute the command from right-clicking on the, 'Execute' menu item from the command list.

SetSessionEnabled(Guid *InternalID, Boolean Enabled*) – This function works exactly like the one above except that the command to be affected is located by its InternalID.

GetSessionEnabled(string *CommandName*) - This Boolean function returns the enabled state of a command indicated by *CommandName*. This function returns true if the indicated command is enabled during the current session and false if the command is temporarily disabled (see, 'SetSessionEnabled' above). The command is located by its name specified in *CommandName*.

GetSessionEnabled(Guid *InternalID*) - This Boolean function works exactly like, 'GetSessionEnabled' above, except the command is located by its InternalID.

SetSessionEnabledByCategory(string *CategoryName, Boolean Enabled*) - This function temporarily sets the enabled state of a set of commands (indicated by category name) during the current session (that is, while VoiceAttack is running – the setting is not saved). If a command is able to be executed (by voice, keyboard shortcut, mouse button, etc.), setting *Enabled* to false will temporarily disable the all commands of a given category from

executing. Setting *Enabled* to true again will re-enable the commands. Important: If you edit the category of a command affected by this function, you will need to re-execute this function in order for the changes to be effective.
Note: You will still be able to execute the affected commands from right-clicking on the, 'Execute' menu item from the command list.

GetSessionEnabledByCategory(string *CategoryName*) - This Boolean function returns true if the indicated category is enabled and false if the category has been set to be temporarily disabled during the current session (see, 'SetSessionEnabledByCategory' above).

Execute (string *CommandPhrase,* optional Boolean *WaitForReturn,* optional Boolean *AsSubcommand,* optional Action<Guid?> *CompletedAction,* optional *PassedText*, optional *PassedIntegers*, optional *PassedDecimals*, optional *PassedBooleans*, optional *PassedDates*) - This method will execute a VoiceAttack command in the active profile by the spoken phrase indicated in *CommandPhrase*. The flow of execution will continue immediately if you pass in false (the default) for *WaitForReturn*. If you want the flow of execution to wait until the command completes, pass in true. Pass a true value as the *AsSubcommand* parameter to execute the command in the context of a subcommand to the calling command. This will allow the executed command to gain certain attributes of the calling command, such as command-shared variables and also allows the command to execute if the calling command is called synchronously. The optional parameter *CompletedAction* will allow you to specify a function that is called when the command completes execution (that is, after all of its actions have been invoked). What is returned to the function's single nullable GUID parameter will be the internal id of the command.

Optional parameters *PassedText*, *PassedIntegers*, *PassedDecimals*, *PassedBooleans and PassedDates* are all string variables that you can use to pass different types of values to the executed command. If you are a programmer, this is most akin to passing in parameters to a function. The executed command will convert any passed values to command-scoped variables that are accessible only within the executed command. *** In order to save space (and to hopefully save some trees), this whole feature is explained earlier in the, 'Execute Another Command' action section. Just search for, 'Passed Values (Advanced)'. ***

Notes - If the command does not exist, the log will display an alert. If you execute a command using this method within a proxy or plugin initialization method or by clicking the, 'Test' button of the Inline Function editor, *AsSubcommand* will be ignored, since the execution will not be within the context of an executing command.

```
vaProxy.Command.Execute("fire weapons"); //continues immediately (asynchronous)
vaProxy.Command.Execute ("fire weapons", true); //waits until the command com-
pletes
vaProxy.Command.Execute ("fire weapons", true, true); //waits until the command
completes and also execute the command as a subcommand


//A quick example of how to use the, 'CompletedAction' parameter.
//First, we must set up the callback function (note the single parameter):
```

```
public void MyCompletedFunction(Guid? theInternalId)
{
   VA.WriteToLog(theInternalId.ToString() + “ completed”, “Orange”);
}

vaProxy.Command.Execute (“fire weapons”, true, true, MyCompletedFunction);
//this waits until the command completes, executes the command as a subcommand and
//executes, ‘MyCompletedFunction’ when it is finished.

//Here is an example of executing a command and passing two text variables, one
//integer literal and one Boolean value that is to be resolved from a token:
vaProxy.Command.Execute(“fire weapons”, true, false, null, “myVar1;myVar2”, “55”,
null, “{TXTRANDOM:true;false}”);

//same example, using named parameters (since all these optional parameters can be
//a little overwhelming):
vaProxy.Command.Execute(“fire weapons”, WaitForReturn: true, PassedText:
“myVar1;myVar2”, PassedIntegers: “55”, PassedBooleans: “{TXTRANDOM:true;false}”);
```

Execute(Guid *InternalID,* optional Boolean *WaitForReturn,* optional Boolean *AsSubcommand,* optional Action<Guid?> *CompletedAction*) - This method works exactly like the Execute method above, except this version locates the command to execute by InternalID.

Exists(string *CommandPhrase*) – This Boolean function returns true if a command is available to the active profile with the spoken phrase specified in *CommandPhrase*.

```
if (vaProxy.Command.Exists(“fire weapons”))
{
    vaProxy.Command.Execute(“fire weapons”);
}
```

Exists(Guid *InternalID*) – This Boolean function returns true if a command is available to the active profile with InternalID specified in *InternalID*.

Active(string *CommandPhrase*) - This Boolean function returns true if the command indicated by spoken phrase in *CommandPhrase* is actively executing (most likely long-running or running in a loop).

```
if (vaProxy.Command.Active(“fire weapons”) == false) //if command not active
{
    vaProxy.Command.Execute(“fire weapons”);  //execute the command
}
```

Active(Guid *InternalID*) - This Boolean function returns true if the indicated command is actively executing (most likely long-running or running in a loop). The difference between this function and the one above is that this version locates the command via InternalID and not by name.

ActiveCount(string *CommandPhrase*) - This integer function returns the number of running instances of a command indicated by spoken phrase in *CommandPhrase* (most

likely long-running or running in a loop).

ActiveCount(Guid *InternalID*) - This integer function returns the number of running instances of a command indicated by its InternalID in *InternalID* (most likely long-running or running in a loop).

AlreadyExecuting() - This Boolean function returns true if the currently-executing command is actively executing in another instance (most likely long-running or running in a loop).

CategoryExists(string *CategoryName*) – This Boolean function returns true if any command is available to the active profile that has a category with the name specified in *CategoryName*.

LastUserExec() – returns an int that indicates the number of seconds since the last command executed by the user - spoken phrase, keyboard key press, mouse click or joystick button press. That is, it does not include subcommands, commands executed externally, rightclick execute, etc.

ExecutionCount() – returns an int that is the count of top-level commands (commands that are not subcommands) that have executed since VoiceAttack had launched.

LastSpoken() – returns a string which indicates the last-spoken command phrase. Useful from within sub-commands where you need to know what was said to invoke the root command, or if you need to know what was spoken prior to the current command (if the current command was not spoken (executed by keyboard, mouse, joystick, external, etc.).

PreviousSpoken() – returns a string that indicates the spoken command phrase prior to the last spoken command phrase.

SpokenHistory(int *value***)** – returns a string. This provides access to the history of spoken commands (up to a maximum of 1000 phrases), starting at 0. A value of zero (SpokenHistory(0)) is the same as getting the value of the LastSpoken() function. A value of 1 (SpokenHistory(1)) is the same as getting the value of PreviousSpoken(). A value of 2 would get the spoken phrase that was issued before the previous spoken command and so on. If no history exists for the value, a blank (empty string) is returned.

vaProxy.Profile Attributes

Below are the attributes that belong to the Profile object of vaProxy. These attributes pertain to profiles (including the profile that is currently active).

Name() – This function returns the name of the active profile.

```
string profileName = vaProxy.Profile.Name();
```

InternalID() – This function returns a nullable Guid (Guid?) that indicates the InternalID value of the active profile. The profile's InternalID is an author flag (see, 'Author Flags'

section).

Exists(string *ProfileName*) – This Boolean function returns true if a profile is available with the name specified in *ProfileName*.

Exists(Guid *InternalID*) – This Boolean function returns true if a profile is available with the InternalID specified in *InternalID*.

AuthorTag1(), **AuthorTag2()**, **AuthorTag3()** – Returns the value for AuthorTag1, AuthorTag2 and AuthorTag3 (see, 'Author Flags' section).

AuthorID() – Returns the value for AuthorID (see, 'Author Flags' section). The return type for this function is nullable Guid (Guid?).

ProductID() – Returns the value for ProductID (see, 'Author Flags' section). The return type for this function is nullable Guid (Guid?).

PreviousName() – returns a string that indicates the name of the profile that was loaded prior to the currently-loaded profile.

PreviousInternalID() – This function returns a nullable Guid (Guid?) that indicates the InternalID value of the profile that was loaded prior to the currently-loaded profile. A profile's InternalID is an author flag (see, 'Author Flags' section).

PreviousAuthorTag1(), **PreviousAuthorTag2()**, **PreviousAuthorTag3() -** These return the three different AuthorTags of the profile that was loaded prior to the currently-loaded profile (see, 'VoiceAttack Author Flags' later on in this document). This is an advanced feature that will probably not be used by most.

PreviousAuthorID() – returns the value of AuthorID from the profile that was loaded prior to the currently-loaded profile. The return type of this function is nullable Guid (Guid?) (see, 'Author Flags' section).

PreviousProductID() –returns the value of ProductID from the profile that was loaded prior to the currently-loaded profile. The return type of this function is nullable Guid (Guid?) (see, 'Author Flags' section).

NextName() – returns a string that indicates the name of the profile that has been selected to load after the current profile is unloaded. This function will return as empty if the profile is not unloading or if the profile is unloading due to VA shutting down. Note: If you haven't been able to tell by now, this function is only useful from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document).

NextInternalID() – This function returns a nullable Guid (Guid?) that indicates the InternalID value of the profile that has been selected to load after the current profile isi unloaded. InternalID is an author flag (see, 'Author Flags' section). Note: If you haven't been able to tell by now, this function is only useful from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document).

NextAuthorTag1(),**NextAuthorTag2()**, **NextAuthorTag3() -** These return the three different AuthorTag values of the profile that has been selected to load after the current profile is unloaded. These functions will return as empty if the profile is not unloading or if the profile is unloading due to VA shutting down. Note: Just like NextName(), these functions are only useful from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document, as well as the, 'Author Flags' section later in this document). This is an advanced feature that will probably not be used by most.

NextAuthorID() – returns the value of AuthorID from the profile that has been selected to load after the current profile is unloaded. This function will return as null if the profile is not unloading or if the profile is unloading due to VA shutting down. The return type of this function is nullable Guid (Guid?) (see, 'Author Flags' section). Note: If you haven't been able to tell by now, this function is only useful from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document).

NextProductID() – returns the value of ProductID from the profile that has been selected to load after the current profile is unloaded. This function will return as null if the profile is not unloading or if the profile is unloading due to VA shutting down. The return type of this function is nullable Guid (Guid?) (see, 'Author Flags' section). Note: If you haven't been able to tell by now, this function is only useful from within an unload command (see, 'Execute a command each time this profile is unloaded' earlier in this document).

History(int *value***)** – returns a string. This provides access to the history of the names of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero (History(0)) is the same as getting the value of the Name() function. A value of 1 (History(1)) is the same as getting the value of PreviousName(). A value of 2 would get the name of the profile loaded before the previous profile and so on. If no history exists for the value, a blank (empty string) is returned.

HistoryInternalID(int *value***)** – returns a nullable Guid?. This provides access to the history of the InternalID values of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero (HistoryInternalID(0)) is the same as getting the value of the InternalID() function. A value of 1 (HistoryInternalID(1)) is the same as getting the value of PreviousInternalID(). A value of 2 would get the InternalID of the profile loaded before the previous profile and so on. If no history exists for the value, null is returned.

HistoryAuthorID(int *value***)** – returns a nullable Guid (Guid?). This provides access to the history of the AuthorID values of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero (HistoryAuthorID(0)) is the same as getting the value of the AuthorID() function. A value of 1 (AuthorID(1)) is the same as getting the value of PreviousAuthorID(). A value of 2 would get the AuthorID of the profile loaded before the previous profile and so on. If no history exists for the value, null is returned.

HistoryAuthorTag1(int *value***)**, **HistoryAuthorTag2(**int *value***), HistoryAuthorTag3(**int *value***) -** This provides access to the history of the three AuthorTag values of loaded profiles (up to a maximum of 1000 profiles), starting at 0. A value of zero (HistoryAuthorTag1(0)) is the same as getting the value of the AuthorTag1() function. A value of 1 (HistoryAuthorTag1(1)) is the same as getting the value of PreviousAuthorTag1(). A value of 2 would get the Author Tags of the profile loaded before the previous profile and so on.

If no history exists for the value, a blank (empty string) is returned. (See, 'VoiceAttack Author Flags' later on in this document). This is an advanced feature that will probably not be used by most.

SwitchTo(String *ProfileName***)** – This function attempts to switch to the profile indicated by name in *ProfileName*. Note that switching profiles causes the current profile to cease execution.

SwitchTo(Guid *InternalID***)** – This function attempts to switch to the profile indicated by InternalID. Note that switching profiles causes the current profile to cease execution.

Reset() - This function reloads the current, active profile. Note that reloading the current profile will cause the profile's execution to cease.

vaProxy.Queue Attributes

Below are the attributes that belong to the Queue object of vaProxy. These attributes pertain to getting information about command queues that may be active.

Count() – returns an int that indicates the number of command execution queues that are currently available.

```
int iQueueCount = vaProxy.Queue.Count();
```

Status(string *Name***)** – returns a string that is the status of a command execution queue indicated in the Name parameter. The following values will be returned:

'Not Initialized' – This will be the returned status if the queue does not exist (that is, no commands have been enqueued into a queue by the name given). 'Running' - The queue is currently running (not paused) and there is at least one command in the queue. 'Idle' - The queue is running (not paused) but there are zero items in the queue. 'Paused' - The queue is paused, or flagged to be paused if a command is currently executing within the queue. 'Stopped' - The queue is in a stopped state or has been flagged to stop if a command is currently executing within the queue.

```
if (vaProxy.Queue.Status("myQueue") == "Running")
{
    //do something here
}
```

CommandCount(string *Name***)** – returns an int that is the number of commands contained within a command execution queue that is indicated in the Name parameter. If the queue does not exist, the return value will be 0.

ActiveCommandName(string *Name***)** – returns a string that is the name of the executing command of a command execution queue (indicated in the Name parameter). If the queue does not exist, or if there is no command currently executing within the queue, an empty

value ‘’ will be returned.

vaProxy.State Attributes

Below are the attributes that belong to the State object of vaProxy. This set of functions deals with various states of devices, your computer, and VoiceAttack itself. Just like the, ‘Utility’ object earlier in this section, this is a bunch of functions that VoiceAttack uses internally, but is exposed to hopefully make things a bit easier for you for certain stuff.

GetListeningEnabled() - This Boolean function returns true if VoiceAttack’s, ‘listening’ is turned on, false if it is not.

SetListeningEnabled(bool *Value*) - This function turns VoiceAttack’s, ‘listening’ on and off. Passing a true value turns, ‘listening’ on, while passing a false value turns, ‘listening’ off.

GetShortcutsEnabled() - This Boolean function returns true if VoiceAttack’s keyboard shortcut keys are enabled.

SetShortcutsEnabled(boo*l Value*) - This function turns VoiceAttack’s keyboard shortcuts on and off. Passing a true value turns the shortcuts on, while passing a false value turns them off.

GetJoystickButtonsEnabled() - This Boolean function returns true if VoiceAttack’s joystick shortcut buttons are enabled.

SetJoystickButtonsEnabled(*bool Value*) - This function turns VoiceAttack’s joystick shortcuts on and off. Passing a true value turns the shortcuts on, while passing a false value turns them off.

GetMouseButtonsEnabled() - This Boolean function returns true if VoiceAttack’s mouse shortcut buttons are enabled.

SetMouseButtonsEnabled(*bool Value*) - This function turns VoiceAttack’s mouse shortcuts on and off. Passing a true value turns the shortcuts on, while passing a false value turns them off.

KeyDown(String *Key***)** – returns a Boolean value of true if the indicated keyboard key is pressed down. The, 'Key' parameter can be any key you can type in a token: 'A', 'B', 'C', 'ß', 'ö', 'ñ', 'ç', as well as keys you can't type in: ENTER, TAB, LCTRL, ARROWR (see section later in this document titled, **'Key State Token Parameter Values'** for the full list). For example, if you want to test to see if the F10 key is down, just call KeyDown(“F10”). To test for the letter 'A', just call KeyDown(“A”).

AnyKeyDown() – returns a Boolean value of true if any keyboard key is currently in the pressed down state, and returns false if no keys are pressed down.

MouseLeftButtonDown()
MouseRightButtonDown()

MouseMiddleButtonDown()
MouseForwardButtonDown()
MouseBackButtonDown() - Each of these functions test to see if a mouse button is being pressed. For example, if you want to test for the right mouse button, use the function MouseRightButtonDown().. If the mouse button is pressed down, the returned value will be true, otherwise it will be false.

AnyMouseButtonDown() - This Boolean function checks to see if any of the five standard mouse buttons are currently pressed. If any mouse button is down, the returned value will be true. If no buttons are pressed, false will be returned.

CPU(int *Core***)**
CPU() - These will return your cpu usage as an integer. CPU() will return the average for all cores. The value returned will be from 0 to 100. **CPU(**int *Core***)** will allow you to specify a particular core. For instance, CPU(5) will get the cpu usage for core 5.

RAMTotal() - This will return the total RAM on your system in bytes as an unsigned long (ulong).

RAMAvailable() - This will return the available RAM on your system in bytes as an unsigned long (ulong).

FileExists(String *Path***) -** This Boolean function will return true if the file indicated in Path exists, or false if it does not.

DirectoryExists(String *Path***) -** This Boolean function will return true if the directory indicated in Path exists, or false if it does not.

DirectoryHasFiles(String *Path***) -** This Boolean function will return true if the directory indicated in Path has files in it, or false if it does not. Note that if the directory does not exist, false will also be returned.

AudioLevel() - This function indicates the currently reported audio level from the speech engine as an integer. The returned value will be from 0 to 100.

AudioLastFile() - This will return the path of the last audio file that is played as a string.

AudioCount() - This returns the number of all currently-playing audio files as an integer. If legacy audio mode is on this value will always be zero.

AudioCount(String *Path***) -** This returns the number of currently-playing instances of an audio file with a given file path as an integer.
Notes: Since sounds run asynchronously in VoiceAttack, there is a slight chance that if you use this function *IMMEDIATELY* after executing a 'Play a Sound' action the file may not yet have had a chance to queue or load up and will not be included in the count. This is technically correct, but may not be a proper count depending on what you are trying to accomplish.

If legacy audio mode is on this function will always return zero.

AudioPos(String *Path***)** - This returns the position of currently-playing audio file with a given file path, expressed in seconds as an integer value.
Notes: Since sounds run asynchronously in VoiceAttack, there is a slight chance that if you use this token *IMMEDIATELY* after executing a 'Play a Sound' action the file may not yet have had a chance to queue or load up and will return a position of 0. This is technically correct, but may not be a proper value depending on what you are trying to accomplish.

Since you can run multiple instances of a sound file at once, if there is more than one instance currently playing, the rendered value will be zero, (assumptions cannot be made on which instance to be chosen). To help with this, check the AudioCount() function outlined above prior to using the AudioPos() function.

If no instances of the indicated file are currently playing, zero will be returned.

If legacy audio mode is on this function will always return zero.

AudioOutputType() - This returns the currently-selected audio output type as indicated on the Audio tab of the Options screen. The possible values are, "**Legacy**" when, "Legacy Audio" is selected, "**Windows**" when "Windows Media Components" are selected and, "**Integrated**" when, "Integrated Components" is selected.

DefaultPlayback() - This returns the device name of the default multimedia audio playback device as indicated by Windows as a string. Note that there is a very minor memory leak when accessing the multimedia device property store, so this will be reflected in VoiceAttack (using this function sparingly will not present a problem… running it over and over in a loop will chew up memory… the search for a better way continues).

DefaultPlaybackComms() - This works just like DefaultPlayback() (above), except the default communications playback device name will be returned.

DefaultRecording() - This works just like DefaultPlayback() (above), except the default multimedia recording device name will be returned.

DefaultRecordingComms() - This works just like DefaultPlayback() (above), except the default communications recording device name will be returned.

SysDir() - This returns the path of the system directory (e.g. 'C:\Windows\System32').

WinDir() - This returns the path of the Windows directory (e.g. 'C:\Windows').

EnvironmentVariable(String *Value***)** - This renders the Windows environment variable specified in, 'Value' as a string. For example, EnvironmentVariable("programfiles") would (usually) return 'C:\Program Files'.

Culture() - This returns the default user locale of the system as a string.

UICulture() - This returns the default user interface language as a string.

Is64Bit() - This returns true if the VoiceAttack process is 64-bit. False if it is not.

SysVol() - This returns the system volume (default playback device) as an integer value from 0 to 100.

SysMute() - If the system volume (default playback device) is muted, this function returns true. If not, the value returned is false.

MicVol() - This returns the microphone volume (default recording device) as an integer value from 0 to 100.

MicMute() - If the microphone volume (default recording device) is muted, this function returns true. If not, the value is returned is false.

AppVol(String *Context***)** - This function returns an integer value between 0 and 100 based on the indicated app volume as it relates to the **System Volume Mixer**. If the volume cannot be accessed (app closed or no audio is playing for instance), a value of -1 will be returned. The Context parameter for this function should be the window title, process name or window class name of the target application (see "Set Audio Level" section of the, "Other Stuff" screen for more information on targeting the application).

AppMute(String *Context***) -** This works exactly the same as the AppVol() function above, except this returns a value of true if the indicated application is muted as it relates to the System Volume Mixer, and false if the application is not muted. False is returned if the information cannot be accessed.

SpeechDeviceMute() - This function tests if the recording device that the speech engine is currently using is muted. If the device is muted, a Boolean value of true will be returned. Otherwise, the returned value will be false.

SpeechDeviceVol() - This returns the volume of the recording device that the speech engine is currently using as an integer value from 0 to 100.

SpeechActive() - This function tests to see if the speech engine is detecting speech. If speech is currently detected, the returned value will be true. If, not, the returned value will be false.

State.Joystick – The Joystick object within State holds a collection of methods that are a direct port of all the token functions outlined in the token function section of this help document. Since the audience for these methods will most likely be extremely small, and for brevity (and to hopefully save a tree), the full documentation on these methods has been omitted. Please refer to the section titled, 'Joystick State Token Reference' – the methods are pretty much a 1 to 1 match. If you have questions or need clarification on this object, please come by the VoiceAttack Discord server or the VoiceAttack user forum. If it turns out that these methods are in high demand, proper documentation will be added here.

vaProxy.Dictation Attributes

Below are the attributes that belong to the Dictation object of vaProxy. This set of functions deals with starting dictation, stopping dictation, clearing the dictation buffer and checking the mode. These functions behave similarly to dictation actions.

Start(out String *Message***)** - This Boolean function returns true if VoiceAttack's dictation mode is successfully turned on. Any message that is generated (success or failure) will be passed back through the out parameter, '*Message*'.

Example:

```
String Message;
if (VA.Dictation.ClearBuffer(false, out Message))
{
    if (Message != null) //success clearing buffer.  Write generated message
        VA.WriteToLog(Message, "blue");

    if (VA.Dictation.IsOn())
        VA.WriteToLog("Already listening, captain.", "orange");
    else
    {
        if (VA.Dictation.Start(out Message)) //success – write our own message
            VA.WriteToLog("Dictation has commenced, captain.", "orange");
        else
            VA.WriteToLog(Message, "red"); //error has occurred starting dictation
    }
}
else
    VA.WriteToLog(Message, "red"); //error has occurred when clearing buffer
```

ClearBuffer(bool *OnlyLastStatement,* out String *Message*) - This Boolean function returns true if VoiceAttack is able to clear the dictation buffer and false if VoiceAttack was not able to clear the dictation buffer. Any message that is generated (success or failure) will be returned through the out parameter, '*Message*'. Setting the Boolean parameter, '*OnlyLastStatement*' to true will indicate that only the last statement in the buffer will be cleared instead of the entire buffer. Example above.

Stop() – This function turns VoiceAttack's dictation mode off.

IsOn() – This Boolean function returns true if VoiceAttack's dictation mode is turned on.

vaProxy Events

ProfileChanging(Guid? *FromInternalID*, Guid? *ToInternalID,* String *FromName,* String *ToName***)** – this event is raised when a profile is about to be changed. As a helper, the InternalID and name of both the current profile and the profile that is about to be loaded are included as parameters. Note that the usage for inline functions and plugins varies slightly.

Example for inline function usage:

```
public void main()
{
    VA.ProfileChanging += MyProfileChangingAction;
}

public void MyProfileChangingAction(Guid? currentID, Guid? loadingID,
                                    String currentName, String loadingName)
{
    VA.WriteToLog("Profile changing to " + loadingName, "orange");
}
```

Example for plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
{
    _proxy = vaProxy;
    _proxy.ProfileChanging += new Action<Guid?, Guid?, String, String>(MyPro
                   fileChangingAction);
}

public static void MyProfileChangingAction(Guid? FromInternalID, Guid? ToInternalID,
                                           String FromName, String ToName)
{
    _proxy.WriteToLog("Profile changing to " + ToName, "orange");
}
```

ProfileChanged(Guid? *FromInternalID*, Guid? *ToInternalID,* String *FromName,* String *ToName***)** – this event is raised when a profile has changed. As a helper, the InternalID and name of both the previous profile and the current profile are included as parameters. Note that the usage for inline functions and plugins varies slightly.

Example for inline function usage:

```
public void main()
{
    VA.ProfileChanged += MyProfileChangedAction;
}

public void MyProfileChangedAction(Guid? previousID, Guid? currentID,
                                   String previousName, String currentName)
{
    VA.WriteToLog("Profile changed to " + currentName, "orange");
}
```

Example for plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
```

```
{
    _proxy = vaProxy;
    _proxy.ProfileChanged += new Action<Guid?, Guid?, String, String>(MyPro
                    fileChangedAction);
}

public static void MyProfileChangedAction(Guid? FromInternalID, Guid? ToInternalID,
                                          String FromName, String ToName)
{
    _proxy.WriteToLog("Profile changed to " + ToName, "orange");
}
```

CommandExecuting(Guid? *InternalID*) **-** this event is raised when a command is about to execute. The InternalID (nullable Guid type) of the executing command is included as a parameter. Note that the usage for inline functions and plugins varies slightly. Note also that in order for this event to be raised by a command, the, 'Enable proxy command events' option must be selected. This is an experimental and unsupported feature with a very limited usage scope.

Example for inline function usage:

```
public void main()
{
    VA.CommandExecuting += MyCommandExecutingAction;
}

public void MyCommandExecutingAction(Guid? InternalID)
{
    VA.WriteToLog("Executing Command with id " + InternalID.ToString(), "orange");
}
```

Example for plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
{
    _proxy = vaProxy;
    _proxy.CommandExecuting += new Action<Guid?>(MyCommandExecutingAction);
}

public static void MyCommandExecutingAction(Guid? InternalID)
{
    _proxy.WriteToLog("Command executing with id " + InernalID.ToString(), "orange");
}
```

CommandExecuted(Guid? *InternalID*) **-** this event is raised when a command has completed execution. The InternalID (nullable Guid type) of the executed command is included as a parameter. Note that the usage for inline functions and plugins varies slightly. Note also that in order for this event to be raised by a command, the, 'Enable proxy command events' option must be selected. This is an experimental and unsupported feature with a very limited usage scope.

Example for inline function usage:

```
public void main()
{
    VA.CommandExecuted += MyCommandExecutedAction;
}

public void MyCommandExecutedAction(Guid? InternalID)
{
    VA.WriteToLog("Command with id " + InternalID.ToString() + " completed", "orange");
}
```

Example for plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
{
    _p = vaProxy;
    _p.CommandExecuted += new Action<Guid?>(MyCommandExecutedAction);
}

public static void MyCommandExecutedAction(Guid? InternalID)
{
    _p.WriteToLog("Command with id " + InernalID.ToString() + " completed", "orange");
}
```

CommandsStopped – This event is raised when a, 'Stop All Commands' action is executed or the user presses the, 'Stop Commands' button on the main screen. Use this to receive notification that this event has occurred so you can clean up and exit your inline function (this can replace the Stopped property and ResetStopFlag() function indicated earlier). Note that if this event is used with the, 'Wait for the inline function to complete' option of an inline function, a ten-second grace period is provided in order to allow the inline function to clean up and exit properly before the thread is terminated. Although available and usable within a plugin, this is intended for use within inline functions (due to its nature, VoiceAttack removes this event when commands are stopped). For plugins, use, 'VA_StopCommand' (recommended).

Example for inline function usage:

```
public void main()
{
    VA.CommandsStopped += MyCommandsStoppedAction;
}

public void MyCommandsStoppedAction()
{
    VA.WriteToLog("CommandsStopped Handled", "orange");
}
```

ApplicationFocusChanged(System.Diagnostics.Process *Process*, String *TopmostWindowTitle***)** – this event is raised when application focus is changed in Windows. For instance, if you switch from Notepad to Wordpad as your focused application, this event will be invoked. The event receives two parameters – the Process that has been focused, and the Window title of that process (as a convenience). This works in

conjunction with the Automatic Profile Switching feature within VoiceAttack, so, ‘Automatic Profile Switching’ must be turned on from the Options screen in order for this event to fire. Use the, ‘AutoProfileSwitchingEnabled’ property to test whether or not automatic profile switching is turned on. Note: The polling frequency for application changes is set to 250ms, and the event is fired asynchronously from within VoiceAttack. Note that the usage for inline functions and plugins varies slightly.

Example for inline function usage:

```
public void main()
{
    VA.ApplicationFocusChanged += MyApplicationFocusChangedAction;
}

public void MyApplicationFocusChangedAction(Process pFocus, String title)
{
    VA.WriteToLog(title + " - " + pFocus.ProcessName, "orange");
}
```

Example for plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
{
    _proxy = vaProxy;
    _proxy.ApplicationFocusChanged += new Action<System.Diagnostics.Process,
      String>(MyApplicationFocusChangedAction);
}

public static void MyApplicationFocusChangedAction(System.Diagnostics.Process pFocus,
      String WindowTitle)
{
    _proxy.WriteToLog(WindowTitle + " - " + pFocus.ProcessName, "orange");
}
```

Variable Change Events

These events are raised when a variable’s value is changed. When this type of event is raised, the name of the variable is passed through *Name*, the previous value of the variable is passed through *FromValue* and the current variable value is passed through *ToValue*. As a helper, the *InternalID* of the command that changed the variable’s value is provided (where applicable). Note that the name of the variable **MUST end with ‘#’** in order for these events to be raised. For instance, ‘myVariable**#**’ will be watched for changes (and subsequently raise its corresponding event when its value changes). A variable named ‘myVariable’ (no ‘#’ at the end) will not be watched for changes. See the section labeled, ‘Advanced Variable Control (Scope and Events)’ later in this document for additional information. Note that *FromValue* and *ToValue* will be null if their values are set to, ‘not set’. Note also that these events are highly experimental, unsupported and may change at any time until further notice.

Each variable type (text, integer, decimal, Boolean and date) has its own event associated

with it as shown below:

TextVariableChanged(String *Name*, String *FromValue,* String *ToValue,* Guid? *InternalID***)** – this event is raised when a watched text variable's value is changed.

IntegerVariableChanged(String *Name*, int? *FromValue,* int? *ToValue,* Guid? *InternalID***)** – this event is raised when a watched integer variable's value is changed.

DecimalVariableChanged(String *Name*, dedcimal? *FromValue,* decimal? *ToValue,* Guid? *InternalID***)** – this event is raised when a watched decimal variable's value is changed.

BooleanVariableChanged(String *Name*, Boolean? *FromValue,* Boolean? *ToValue,* Guid? *InternalID***)** – this event is raised when a watched Boolean variable's value is changed.

DateVariableChanged(String *Name*, DateTime? *FromValue,* DateTime? *ToValue,* Guid? *InternalID***)** – this event is raised when a watched date variable's value is changed.

Examples (inline functions and plugins vary slightly):

Inline function usage:

```
public void main()
{
    VA.DecimalVariableChanged += DecimalChanged;
    VA.TextVariableChanged += TextChanged;
    VA.IntegerVariableChanged += IntegerChanged;
    VA.BooleanVariableChanged += BooleanChanged;
    VA.DateVariableChanged += DateChanged;
}

public void DecimalChanged(String name, decimal? dFrom, decimal? dTo, Guid? InternalID)
{
    String sFrom = dFrom.HasValue ? dFrom.ToString() : "NULL";
    String sTo = dTo.HasValue ? dTo.ToString() : "NULL";
    VA.WriteToLog("Text variable [" + name + "] changed from \"" + sFrom + "\" to \"" +
    sTo + "\"", "orange");
}

public void TextChangedAction(String name, String sFrom, String sTo, Guid? InternalID)
{
}

public void IntegerChanged(String name, int? iFrom, int? iTo, Guid? InternalID)
{
}

public void BooleanChanged(String name, Boolean? bFrom, Boolean? bTo, Guid? InternalID)
{
}

public void DateChanged(String name, DateTime? dFrom, DateTime? dTo, Guid? InternalID)
{
}
```

Plugin usage:

```
private static dynamic _proxy;  //note - keeping a reference to the proxy object

public static void VA_Init1(dynamic vaProxy)
{
    _proxy = vaProxy;
    _proxy.DecimalVariableChanged += new Action<String, decimal?, decimal?,
    Guid?>(DecimalChanged);

    _proxy.TextVariableChanged += new Action<String, String, String,
    Guid?>(TextChanged);

    _proxy.IntegerVariableChanged += new Action<String, int?, int?,
    Guid?>(IntegerChanged);

    _proxy.BooleanVariableChanged += new Action<String, Boolean?, Boolean?,
    Guid?>(BooleanChanged);

    _proxy.DateVariableChanged += new Action<String, DateTime?, DateTime?,
    Guid?>(DateChanged);

}

public static void DecimalChanged(String Name, decimal? From, decimal? To,
       Guid? InternalID)
{
    String sFrom = dFrom.HasValue ? dFrom.ToString() : "NULL";
    String sTo = dTo.HasValue ? dTo.ToString() : "NULL";
    _proxy.WriteToLog("Text variable [" + name + "] changed from \"" + sFrom +
    "\" to \"" + sTo + "\"", "orange");
}


public static void TextChanged(String Name, String From, String To, Guid? InternalID)
{
}

public static void IntegerChanged(String Name, int? From, int? To, Guid? InternalID)
{
}

public static void BooleanChanged(String Name, Boolean? From, Boolean? To,
       Guid? InternalID)
{
}

public static void DateChanged(String Name, DateTime? From, DateTime? To,
       Guid? InternalID)
{
}
```

Remember… if you get stuck, drop by the VoiceAttack user forums and go to the plugin boards. Help is always around ;)

Notes on testing your plugin / setup

To debug a VoiceAttack plugin, there are few things you'll need to do.

First, VoiceAttack needs to be able to see your plugin when it runs so it can interact with your plugin. As you read above, that means that the compiled plugin dll needs to go into VoiceAttack's **Apps** folder within a subfolder:

C:\Program Files (x86)\\VoiceAttack\Apps**MyPluginFolder**\myPlugin.dll

To set this up in your project so that the plugin dll is placed in the indicated folder, first go to the plugin's project settings. Next, click on the Build tab and then put the folder name in the Output path box (for this example, that would be
C:\Program Files (x86)\\VoiceAttack\Apps\MyPluginFolder

After you have this set up, you need to now build your plugin once so that the plugin dll appears in that folder. You may have to adjust security settings depending on your machine (that is, you'll get an error indicating that your dll could not be written). To do that, just right click on the new folder you created in the Apps folder, select, 'Properties' then 'Security' and set All Users (or whatever) to 'Full Control' (or however you want to set up your own security settings). On a side note, if security is an issue on your machine it may be necessary to install VoiceAttack in an area of your system that is not Program Files (x86) – just install VoiceAttack in your documents folder or some other place that your user account has full control (Windows is just doing its best to protect your setup).

Now that you've gotten your project built and the plugin dll is in a subfolder of VoiceAttack's Apps folder, you must now open up VoiceAttack and turn on plugin support (which will then require a restart of VoiceAttack). Go back into Options and open up the plugin manager and select the new plugin you just built (for sanity's sake, just make sure it's the only plugin selected – you'll thank yourself later).

Next, open the plugin's project settings, go down to the Debug tab and set the 'Start external program' value to the path of VoiceAttack (usually C:\Program Files (x86)\\VoiceAttack\VoiceAttack.exe – unless you had to move it for security reasons lol).

Set a breakpoint somewhere like in the VA_DisplayName() function just so you can see things are happening.

You can now just, 'run' the plugin project. VoiceAttack will be launched by Visual Studio first and then the plugin's VA_DisplayName() function will be called (and should stop right on your breakpoint).

Note: You can output your values using the, ‘WriteToLog’ function with the vaProxy object: vaProxy.WriteToLog(“some value”, “orange”); The output is token-parsed so you can use any or all of the tokens in there ({SMALL:}, {INT:}, {BOOL:}, {DEC:}, various {DATE:} and {TXT:}). This is handy so you can see things are happening in VoiceAttack (you knew that already for sure ;)).

Notes on referenced assemblies

A lot of times your plugin will want to reference assemblies (.dlls) that are not part of the .Net framework or are not installed in the Global Assembly Cache (GAC). Your plugin will only work if VoiceAttack can actually locate and use your referenced assemblies. In order for VoiceAttack to locate your referenced assemblies, the assemblies must be located in the GAC (of course), in the same folder as your plugin, in the VoiceAttack installation directory (where the VoiceAttack.exe resides), or in the Shared\Assemblies folder located in the VoiceAttack installation directory.

Version 3 and below plugin parameter notes (this is old stuff but still left here as a reference. Note that there have been some updates to this interface to allow access to vaProxy if you still require this interface).

State parameter

The state parameter that is passed in to VA_Invoke1, VA_Exit1 and VA_Init1 is for your own private use within this plugin. No other plugin has access to the values. It serves as a kind of session, so that you can easily maintain information between calls without having to do any kind of persistence. The dictionary is (String, Object) so you can name your values whatever you want, as well as store any kind of value type. Whatever you do to this dictionary will be reflected in VoiceAttack upon return (VA_Invoke1 and VA_Init1 only). So, if you empty this dictionary, VoiceAttack's copy of this dictionary will be emptied. Null values will not be cleaned out. Note: When the state dictionary is initialized for a plugin, there are three key/value pairs that are included for use. The keys are below (the key names are the same as the token values used elsewhere):
VA_DIR : The installation directory of VoiceAttack.
VA_APPS : VoiceAttack apps/plugins directory.
VA_SOUNDS : VoiceAttack sounds directory.
VA_SOUNDS : To allow plugin version v3 and prior to allow access to the new vaProxy object, the VA_PROXY key was added. You can access the VA_PROXY object like so:

```
dynamic vaProxy = state[“VA_PROXY”];
```

See notes above on how to use vaProxy. Note also that VoiceAttack v1.6 or greater will be needed to be able to use this feature.

Again, these values can be erased and manipulated however you want.

SmallIntegerValues (formerly, 'Conditions') parameter

The small integers parameter is a dictionary of (String, nullable Int16 (or short)). These are the same small integers (conditions) you find when using them within the VoiceAttack interface.

To get values from VoiceAttack into the plugin in VA_Invoke1, simply indicate what small integers to pass in the, 'Small Integer Variables' box in the, 'Execute an External Plugin Function' screen. For example, if you want to pass in, 'myValue1' and 'myValue5', just put 'myValue1;myValue5' (no quotes) in the box. VoiceAttack will copy the values into a new list and pass them in to the plugin for you to use. Inside the plugin's function (either VA_Invoke1 or VA_Init1), you can read 'myValue1' or 'myValue5', alter their values (set to null to remove) and/or add more values to the small integers dictionary. Any changes will be reflected in VoiceAttack upon return. To indicate to VoiceAttack that you want to remove a small integer, simply set its value to null (smallIntegerValues[“myConditionName”] = null). On a programming side-note... Since these are dictionaries, it is necessary to check to see if an item exists before doing anything to it... otherwise things come to a quick halt.

Small integers are public and can be accessed from any plugin or any command within the VoiceAttack user interface. That means that anybody can view or modify these values. You can access the values in conditions using the {SMALL:*variableName*}

({COND:*conditionName*} remains for backward compatibility) token in various places in VoiceAttack.

TextValues parameter

The textValues parameter is a dictionary of (String, String). It behaves pretty much exactly like the smallIntegerValues parameter, except the token to access a text value is {TXT:*textVariableName*}, and you pass in values to your plugin via the Text Variables box in the command action.

(and now for some copy/paste/replace… four times :))

IntValues parameter

The intValues parameter is a dictionary of (String, int). It behaves pretty much exactly like the smallIntegerValues parameter, except the token to access a text value is {INT:*variableName*}, and you pass in values to your plugin via the Integer Variables box in the command action.

DecimalValues parameter

The decimalValues parameter is a dictionary of (String, decimal). It behaves pretty much exactly like the smallIntegerValues parameter, except the token to access a text value is {DEC:*variableName*}, and you pass in values to your plugin via the Decimal Variables box in the command action.

BooleanValues parameter

The booleanValues parameter is a dictionary of (String, Boolean). It behaves pretty much exactly like the smallIntegerValues parameter, except the token to access a text value is {BOOL:*variableName*}, and you pass in values to your plugin via the Boolean Variables box in the command action.

DateTimeValues parameter

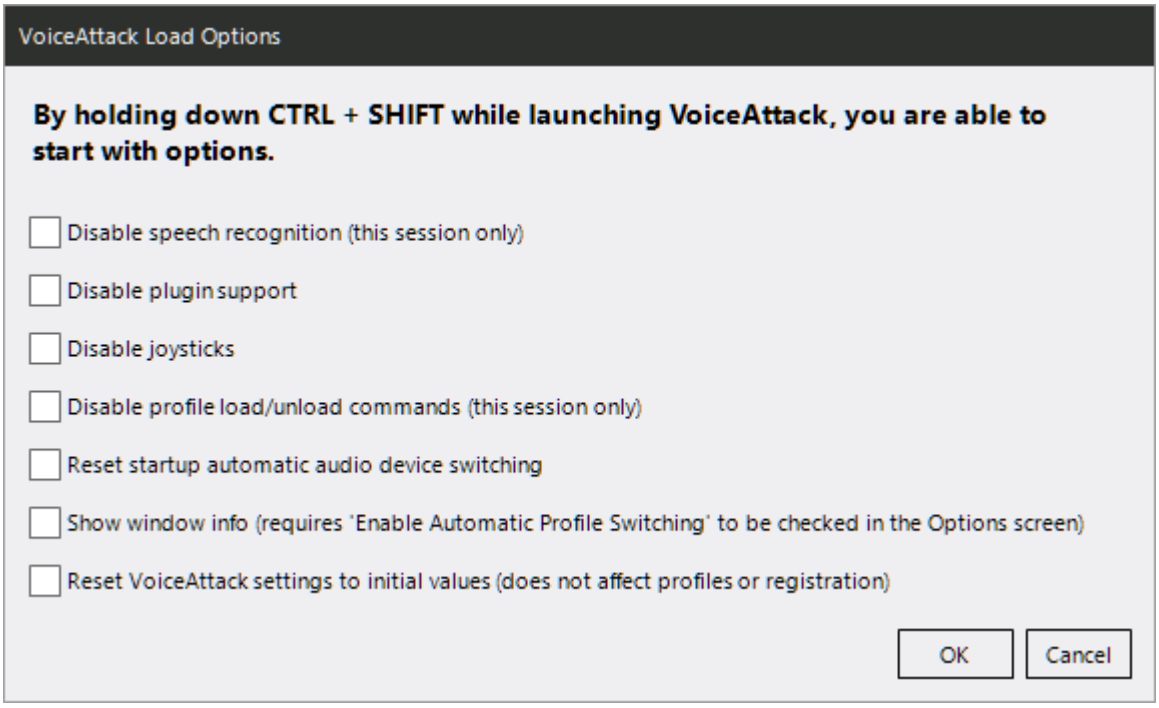
The dateTimeValues parameter is a dictionary of (String, DateTime). It behaves pretty much exactly like the smallIntegerValues parameter, except the token to access a text value is {DATE:*variableName*}, and you pass in values to your plugin via the Date/Time Variables box in the command action.

Context parameter (VA_Invoke1 only)

This parameter is just a string that can be used to get a value into the plugin. This was added so that you don't have to go through the legwork of setting a condition or textValue just to get something simple passed. Convert the string value inside the plugin to whatever type you need to use. You will probably use this parameter the most.

VoiceAttack Load Options Screen

Sometimes you may need to change a particular setting before VoiceAttack loads. If you hold down Control + Shift before and during VoiceAttack's launch, you will get the following screen:



From here, you can disable the speech engine's recognition facilities. This is available in case your speech engine is not working and you're not able to get into VoiceAttack for some reason. Note this option is for the current session only and the next time you run VoiceAttack, the speech engine will be enabled (that is, unless you select the, 'Disable Speech Recognition' on the System/Advanced tab of the Options screen).

You can disable plugin support (in case you get a rogue plugin that you don't want launched on startup). Setting this will disable plugin support until you enable it again in the Options screen.

You can also disable joystick support. This will disable joystick support until you enable it again.

Disabling profile load/unload commands will prevent any of the commands you have chosen to execute on profile change (both loading and unloading) from actually executing. Note that this only occurs on the current running instance of VoiceAttack. If you run VoiceAttack again without this setting, the profile initialization commands will be enabled again.

'Reset startup automatic audio device switching' will reset the options that allow for playback and recording devices to be automatically set on startup. This will help if VoiceAttack is experiencing device trouble accessing these devices.

Selecting, 'Show window info' will display the window title and process name from selected foreground windows. This is so you can easily see what VoiceAttack sees when deciding how to choose your target applications for automatic profile switching. Note that this only occurs on the current running instance of VoiceAttack.
Also Note: This option will **only** work if the, 'Enable Automatic Profile Switching' option is turned on in the main Options screen.

Checking the, 'Reset VoiceAttack settings to initial values' box will wipe out your VoiceAttack user settings (just like clicking the, 'Reset Defaults' button on the Options screen). Note that your profiles and registration information are preserved.

Once you have made your selections, just click, 'OK' and VoiceAttack will continue to load.

NOTE: This screen will automatically appear if VoiceAttack is not shut down properly (probably due to a crash). You can choose to not automatically show this screen from the dialog that pops up prior to the Load Options screen being displayed.

Advanced Variable Control (Scope and Events)

This section is to provide a little bit of guidance on variable, ‘scope’ in VoiceAttack. Initially, all variables in VoiceAttack were globally-scoped. That means that when a variable is set, it is available to be read from and written to from any command in any profile. For the most part, this system works well and will continue to be used. The only gotcha is that global variables must be named very uniquely in order to not interfere with each other. For example:

You have two profiles: Profile A and Profile B. Profile A has a command that creates a text variable, ‘myTextVariable’. All of Profile A’s commands can see and modify, ‘myTextVariable’. Also, when you switch to Profile B, all of Profile B’s commands can see and modify the same, ‘myTextVariable’ variable. In most situations this is alright, but let’s say your friend created Profile B and your friend also uses a text variable called, ‘myTextVariable’. If you switch between Profile A and Profile B, you can see where modifying the same variable can be a problem. Profile A and Profile B would need to be very aware of each other’s variables (and that’s not much fun).

In v1.6.2, some additional control was introduced to make variables a little more private and more contained. Variables can now also be scoped to stay within the profile (profile-scoped) as well as within individual commands (command-scoped).

In order to not triple up the user interfaces for variables, variable names will now have a prefix associated with them. Profile-scoped variables will be prefixed with the, '**>**' (greater-than) character, and command-level variables will be prefixed with, '**~**' (tilde) character. Global variables will remain un-prefixed. So, 'myTextVariable' will be globally-scoped, '**>**myTextVariable' will be profile-scoped and '**~**myTextVariable' will be scoped at the command level.

Using the previous example for profile-scoped variables, if Profile A and Profile B *both* happen to have a variable called, ‘>myTextVariable’ (note the, ‘>’ character), each profile will have its own copy of ‘>myTextVariable’. So, if, ‘>myTextVariable’ in Profile A is set to ‘hello’, ‘>myTextVariable’ in Profile B can be set independently to, ‘world’.

Command-scoped variables work the same way, except that a command-scoped variable is only available to the command instance in which it is executing. It is not available to any other executing commands that may be running, even if those commands are additional instances of the same command. So, if Command A is executing, Command B cannot read or write any of Command A’s command-scoped variables. Additionally, if Command A is run several times in parallel (asynchronously), each running instance of Command A will have its own set of command-scoped variables.

More advanced scope stuff:
If you are still with me down here, there are two more additional prefixes that you can use. First, when a profile is switched, the profile-scoped variables are lost. If you want to retain a profile-scoped variable between profile changes, prefix your variables with two '>' characters. For example, '>myTextVariable' (single, '>') will be lost on profile switch. '>>myTextVariable' (note the double '>') will be retained and will be available when you switch back to that profile.

The last prefix (and my favorite) is the command-shared scope prefix. Command-scoped

variable names that are prefixed with two '~' characters are considered command-shared. That means that the variable will be, 'shared' among commands that are executing in the same execution chain. Basically, the values are passed from one command to any subcommands and then back when control is released from the subcommands. Clear as mud?

Some examples... First, WITHOUT command-shared variables:
Command A sets text variable ~myText to 'hi'. Command A then executes Command B as a subcommand. Command B reads ~myText and it is, 'Not set' (null). That is because ~myText is command-scoped (single, '~'). It is only available to Command A.

WITH command-shared variables:
Command A sets a shared text variable, '~~myTextVariable' (two, '~') value to 'hello'. Command A then executes subcommand Command B. Since ~~myTextVariable is shared, Command B can then read and write ~~myTextVariable. Command B reads ~~myTextVariable as, 'hello' and then decides to set ~~myTextVariable to, 'greetings'. When Command B completes its execution and control returns to Command A, Command A will read ~~myTextVariable updated as 'greetings'.

Another example: Command A executes Command B. Command B executes Command C. Command C initiates the creation of ~~myTextVariable and sets its value to 'welcome'. When Command C completes, it releases control to Command B. Command B completes and releases control to Command A. Command A reads ~~myTextVariable as, 'welcome'. The idea here is that even though the variable was created two levels down from Command A, it is still accessible to Command A (see note below).

NOTE: All command-shared variables are passed **down** to subcommands. If you want updated values to be passed **back up** to the calling command, the, 'Wait until this command completes before continuing' option MUST be selected for the subcommand. Scoped variables are also accessible via plugins and also work when you save variables to the profile (scope is restored when the variable is retrieved). Note also that enqueued commands will receive command-shared variables passed like a subcommand, however, values cannot be passed up to the calling command.

What makes this my favorite is that this is kind of an unorthodox way to be able to pass private information between commands without having to use global variables or establish parameters or return values. Please make sure to visit the VoiceAttack user forums to talk more about this if you are stumped ;)

Variable Change Events

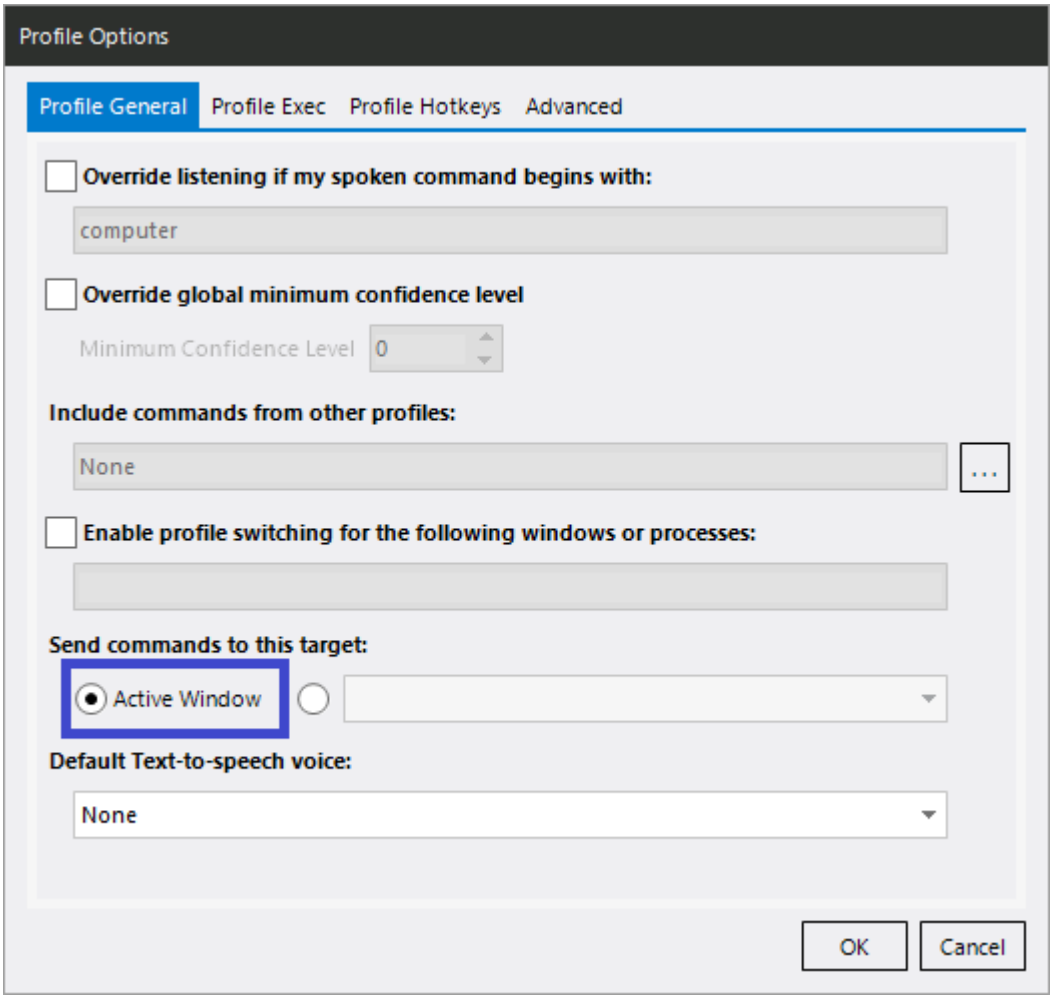
If you have ever built a VoiceAttack plugin or inline function, you've more than likely had situations where you are looping continuously to check if a variable's value has changed. Situations like this can be solved by setting up a variable change event in your inline function or plugin (see **TextVariableChanged**, **IntegerVariableChanged**, **DecimalVariableChanged**, **BooleanVariableChanged** and **DateVariableChanged** events in the 'VoiceAttack Plugins (for the truly mad)' section earlier in this document for more information on how to implement variable change events inside plugins and inline functions). In order to decrease a lot of

overhead, the **only** variables that will be watched for changes (and subsequently raise an event when changed) are variables with names that end with '#'. For example, 'myTextVariable**#**' will be watched for value changes. A variable named 'myTextVariable' will **NOT** be watched for changes. Note that setting a variables value to the same value will not raise an event (since it's not a change lol). Also note that setting a variables value to, 'not set' (or null) if the variable does not already have a value will not raise an event.

Application Focus (Process Target) Guide

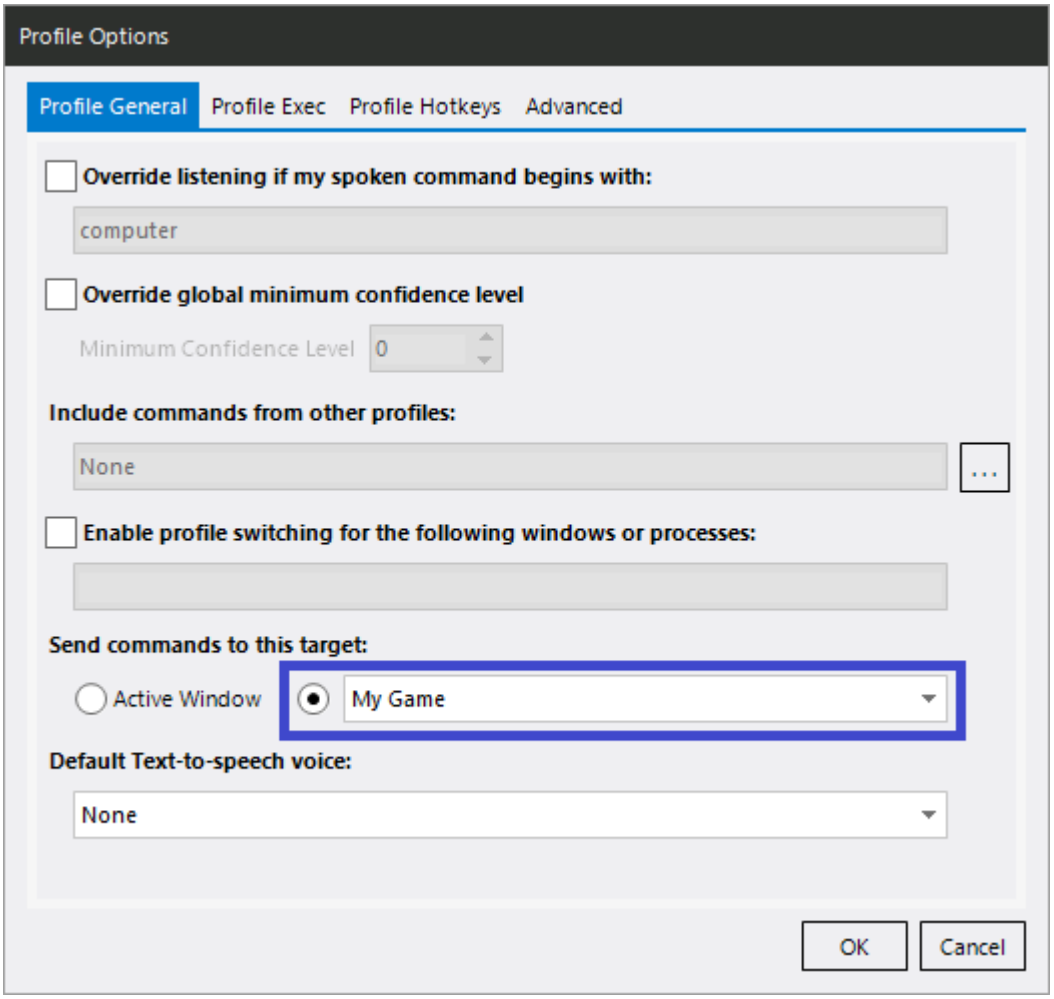
Note: I thought it might be good to consolidate things a bit to get a better picture of what is going on and to illustrate what is available in regards to process targets. In previous versions of VoiceAttack, you had less control over process targets or needed to know a LOT about VoiceAttack in order to change them (like creating sub-commands... bleh). Hope I don't confuse things even more...

Something you'll notice in Windows is that you need to have a window focused in order for it to receive input. With VoiceAttack, the two forms of input are by keyboard and by mouse. In order to type into, say, a comment box in a form or click an, 'OK' button with the mouse, the containing window needs to be at the top of all other windows (foreground) as well as selected (active). If the intended window is not in the foreground and active, nothing will happen. The input that was supposed to go to that window will be occurring in some other window (the window that is actually selected, and in the foreground). In VoiceAttack, the active window in the foreground is called the, 'process target' (or just, 'target').



With all this focusing that needs to be done, VoiceAttack supplies a few ways to help out. The first way is to just use what is called, 'Active Window' as the process target. All profiles use, ‘Active Window’ by default. The profile setting for this is located on the Profile Options screen (see above), which is accessible from the Edit Profile screen.

When, 'Active Window' is selected as the process target, whatever application window that is in the foreground and active at the beginning of the executed command will remain VoiceAttack's target for the duration of the command. So, if Notepad is active in the foreground and a command is executed, Notepad will become the target and any key presses or mouse clicks will be sent to it until the command finishes. If you execute the same command and Wordpad is in the foreground and active, all the same actions will occur against Wordpad. It's a very simple system and it is what is used by most (which is why it is the default setting).

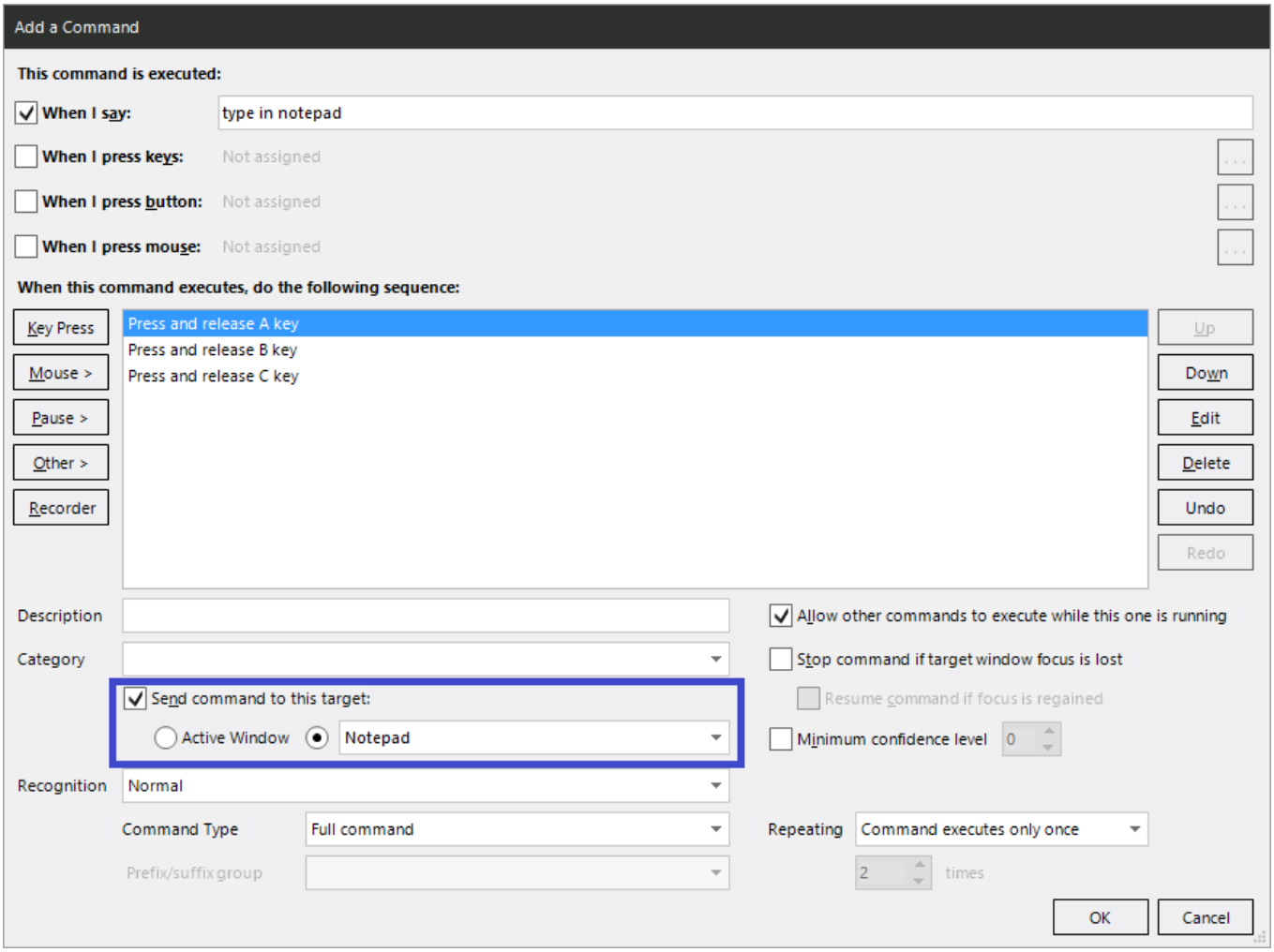


The second way of targeting a window is by selecting the target by name. Targeting by name will make VoiceAttack seek out the window with the specified name and make it the target (that is, bring it to the foreground and activate it) before sending any input to it. So, if you have an application with a window titled, 'My Game', you can have VoiceAttack target the, 'My Game' window by simply selecting the radio button next to the dropdown box, and putting the term, 'My Game' in that dropdown box (as seen above). Note that you can freely type in this box, as well as select applications from the list.

To keep targeting flexible, targets can be selected from the profile options screen of any profile (profile-level) or from any command within a profile (command-level). The target is selected in a cascading manner, which means the command-level target will override

anything specified at the profile level.

Here's an example. Maybe you want, 'My Game' to be the preferred targeting method for your profile (that is, your profile is built specifically for, 'My Game' and pretty much everything in the profile interacts with, 'My Game'), but you want a certain command in this profile to also be able to type some characters into Notepad. Since, 'My Game' is the process target for the entire profile, you will need to be able to specify a process target just for the command (which overrides the profile's setting). You can do this by editing the desired command, selecting the, 'Send command to this target' option, and putting, 'Notepad' in the dropdown box of that option:



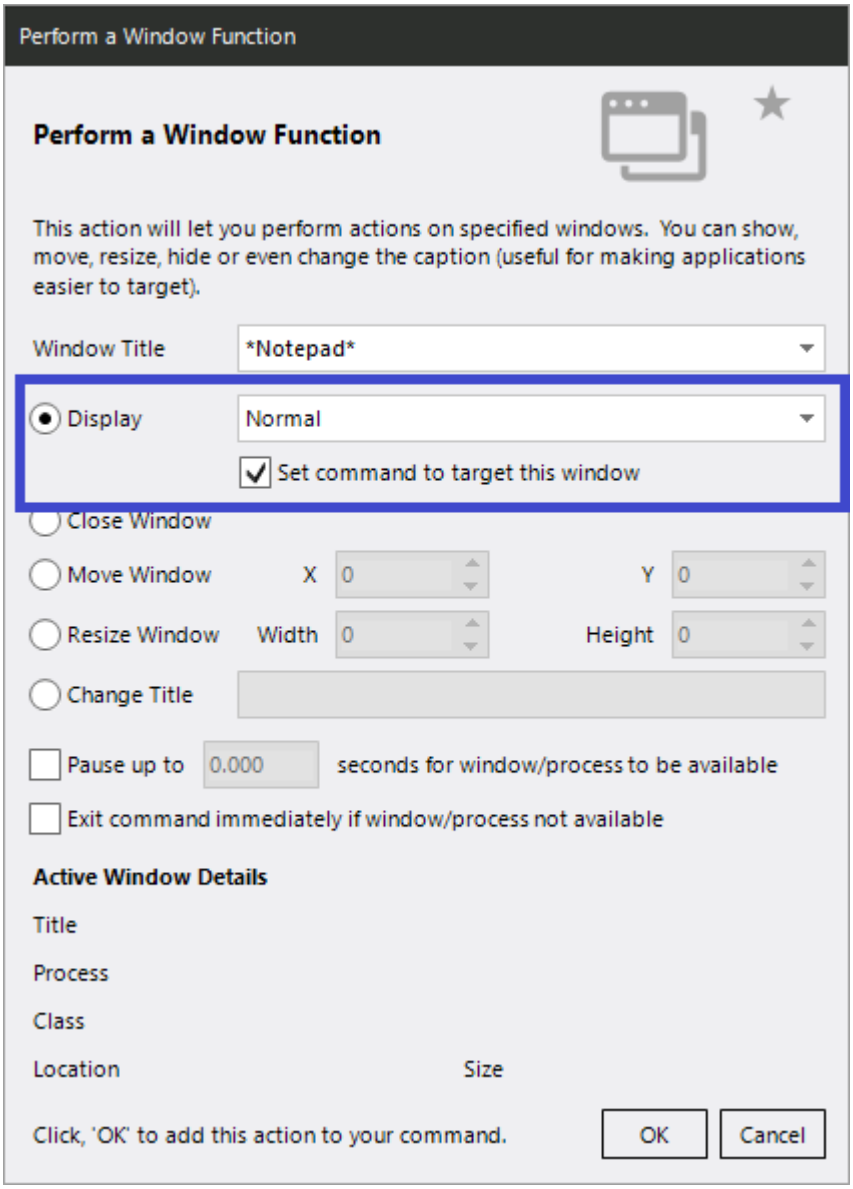
Note: If your window title is the type that changes, you can even specify wildcards in the process target box to make finding the window a little easier by only using a portion of the title (search for, 'wildcard' in this document for more information). You can even change the window's title completely if you need more control (see, 'Perform a Window Function' action for more info). Also, if you do not have the window title handy, you can indicate the process name as it is displayed in Windows Task Manager. The wildcards apply here as well. If the window title and/or process name do not suffice, you can supply the window class name. This is a super-advanced feature and will require you to know the class name for the window. You will need a third-party tool such as Spy++ to get this information, or

observing the class name from the bottom of the, ‘Perform a Window Function’ screen. Wildcards do apply for the window class name if you need them. The window title is searched for first, then the process name and then the window class name.

Advanced targeting

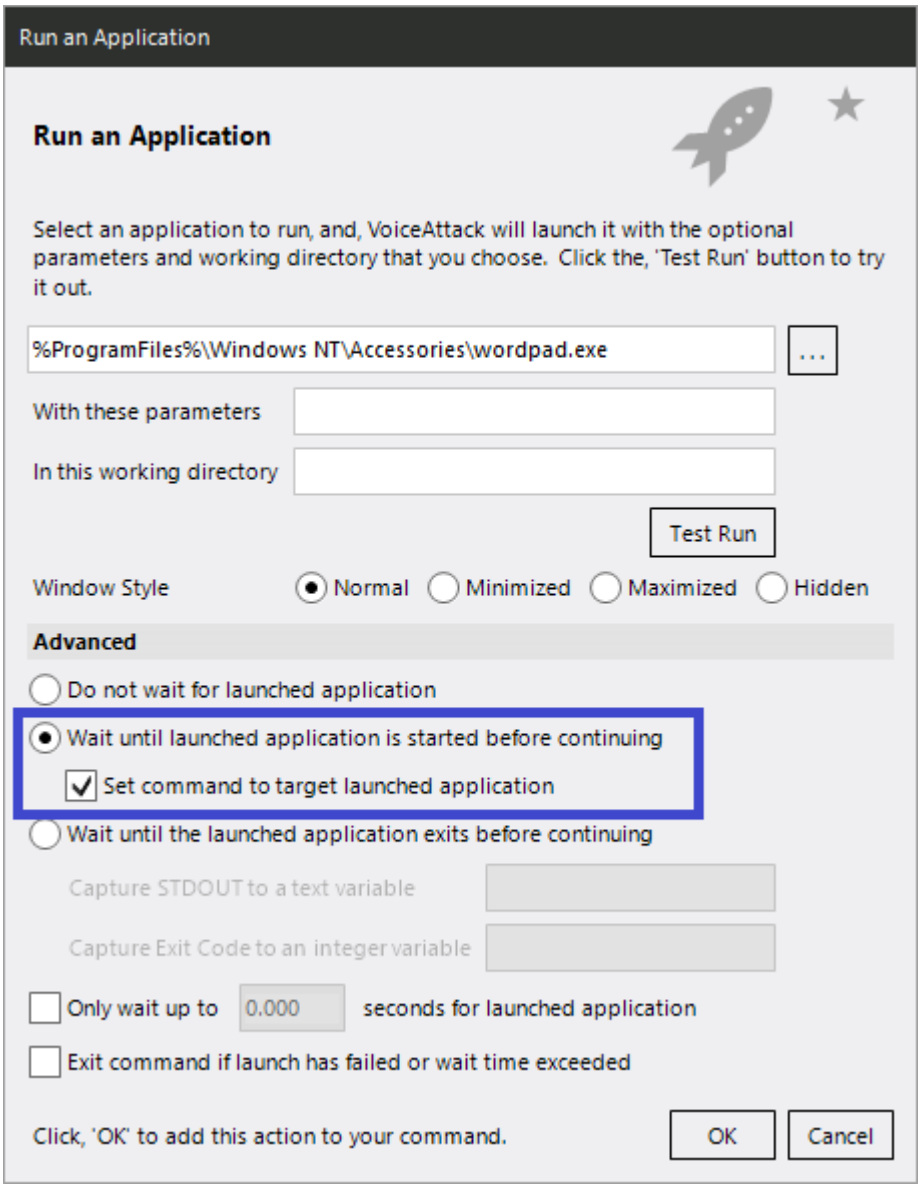
For the most part, the methods above will fill the needs of almost all command macros. The only drawback is that the targeting is too rigid for some situations. For example, when targeted by, 'Active Window', a window remains the target throughout the duration of the command, EVEN IF you manually click another window and make it the foreground window (VoiceAttack will change the process target window BACK to the foreground, selected window as it continues through the command). This is so that activities can occur with other windows, but any input commands (such as pressing keys or typing a message) will continue to be sent to the indicated process target. Similarly, selecting a target by name will only seek out the named window for the entire command. Sometimes there is the need to change the process target even in the middle of a command while it executes. You may need to have a specified target at the start of a command just to get a handle on it, and then redirect input to another application. There may also be the need to launch an application and immediately start sending input to it even if you don't know what it's window title will be. The good news is that VoiceAttack has some features built in to help you handle these types of situations.

The new process target may need to be launched, or the new target may already be running. If the new target is **already running**, you can change to this target by using the, 'Display' feature in the, 'Perform a Window Function' action and selecting the, 'Set command to target this window' option:



What will happen when this action is run is if the target window is found, it will then be set as the process target for the remaining duration of the command. Note that you can set how long to wait for the new target to be active, as well as exit the command completely if the wait time is exceeded (or if the window just simply can't be found).

If the new target **needs to be launched**, just use the 'Run an Application' action and select the, 'Wait until launched application is started before continuing' feature with the, 'Set command to target launched application' option selected:



What will happen is when the application is launched, VoiceAttack will wait until the app is ready to accept input and then set that application as the new process target for the remaining duration of the command. Note that you can also specify how long to wait and to exit if launching the app exceeds the indicated time (or if the launched app fails to launch).

The last way to set a process target happens automatically whenever VoiceAttack uses the mouse to, 'click' on a window (including the desktop). Whenever a mouse button down event occurs (mouse down, click, double-click), the window that is located under the mouse will become the process target (if it is not already) for the remaining duration of the command. This is to simulate what happens when a hardware mouse is clicked. To turn this functionality off and go back to what was in place before, simply select the, 'Bypass

Mouse Targeting' option on the options screen.

Targeting helpers

Commands that are executing are aware of their process target. There are couple of options to note on the command screen: 'Stop command if target window focus is lost' and, 'Resume command if focus is regained'. 'Stop command if target window focus is lost' will stop the command completely if the process target's focus is lost. If you select, 'Resume command if focus is regained', the command will be paused until the process target is active again.

A related feature that has the potential to change the process target is the automatic profile switching feature. We won’t go into any detail here about that, but it's worth mentioning.

Command Execution Queues Overview

Command execution queues (or just, 'queues') execute commands in the order that you add (enqueue) them, making sure that each command is fully executed before moving on to the next command. This will help you execute commands one after the next without having to do all the monitoring yourself. This will also add just a little bit more flexibility in the way that commands can be handled when executed. Think of queues like you would checkout lanes at the grocery store. If a customer (command) gets in line (enqueued), the customer must wait until the person in front of them is finished before checking out (executing). Each subsequent customer must wait until the customer in front of them is finished before being checked out. The lane can fill up and have any number of customers, but, each customer must wait their turn in the order that they showed up.

Let's say you are firing various weapons from your ship, but those operations must occur one after the next (never at the same time for various reasons). Let's say you'd like to first fire your missiles, followed by the rail gun, then machine guns. Normally, if you issued commands for all three of these things in rapid succession, they would all execute at pretty much the same time. If each command is brief, that is usually not much of a problem, but if the commands take a certain amount of time or depend on one another to execute in order, then, that can be an issue. If you enqueue each of these operations - first enqueue the command to fire the missiles, then enqueue the command to fire the rail gun and finally enqueue the command to fire the machine gun - each operation will wait for the last to finish before executing, no matter how long each takes to process.

So, what if you don't want your queued shield commands to wait for your weapon commands to finish? Queues can be named, so that you can have as many queues as you would like to use for different purposes, even at the same time. So, if you have a particular order that you would like to fire weapons (like above), you can create a queue named, 'weapons' and add each weapon command to that queue. You can then create a queue named, 'shields' and then add shield-related commands to that queue. Each queue will execute its commands independently and in the order the commands were added. Note that once a queue is created, it will stay resident until VoiceAttack is closed.

Other neat things that queues can do is that queues can be preloaded with commands to be executed once the queue is filled how you like it, or, the queue's commands can start execution immediately. Queues can also be paused so that once a command is completed, the next command's execution will be postponed until you would like the queue to continue. Queues can also remain running even after they are empty so that any time a new command is added, that command will immediately execute. There's a good bit you can do, and it may seem a little confusing, so, let's walk through creating a queue so you can see how it works.

Creating a Command Execution Queue

You create a command execution queue just by simply enqueueing a command. When you enqueue a command, it's pretty much the same as the old, 'Execute Another Command' action that you've been using for some time. The difference is that when you enqueue a command, instead of the command immediately executing, the command is handed off to a queue that manages that command and any subsequent command that is enqueued. To enqueue a

command, simply add an, 'Enqueue a Command' action by clicking, Other > VoiceAttack Action > Command Queues > Enqueue a Command from the Command screen. From this screen, you'll first want to indicate the name of the queue in which you would like to add the command. Simply type in a name, or, if you've already set up a queue elsewhere, you can select it from this list. The reason that you would want to name your queue is so that you can have more than one active queue. So, if you have a particular order that you would like to fire weapons, you can create a queue named, 'weapons' and add each weapon command to that queue. You can then create a queue named, 'shields' and then add shield-related commands to that queue. Each queue will execute its commands independently and in the order the commands were added.

Next, you will want to indicate whether or not the command is to start executing the moment it is added. If you want the command to start executing immediately, select the, 'Start queue when this command is added' option. If this option is not selected, the command is added to the queue, but queue is not started until you explicitly start it (or, enqueue a subsequent command with this option selected). Your queue can also be started explicitly by executing a, 'Queue Start' action (more about that down below). Once started, your queue will keep waiting for you to add more commands to it. If no commands are in the queue, the next added command will run immediately. If a command is still running in the queue, the latest enqueued command will have to wait until the executing command is finished as well as any other waiting command.

The Queue Actions

In most cases, your command execution queue will be started up and then can just be forgotten. Sometimes you may want to have a little bit of extra control over your queues. There are a few actions that you can perform on a command execution queue, such as starting, stopping and pausing. Note that you can indicate a specific queue or affect all queue instances. The actions are outlined below (you can find more detailed descriptions of each in the, ‘Command Queues - Queue Action’ section earlier in this document):

Start - This will make your queue start executing commands in the order that they were added.

Pause – Pauses the queue once the currently-executing command has completed.

Unpause – Unpauses the queue (starts command execution).

Toggle pause/unpause

Stop – Stops and clears the queue, also stopping any command that may be executing within the queue.

Stop, but allow current command to complete - This will do everything the Stop action will do, except the current command will be allowed to complete.

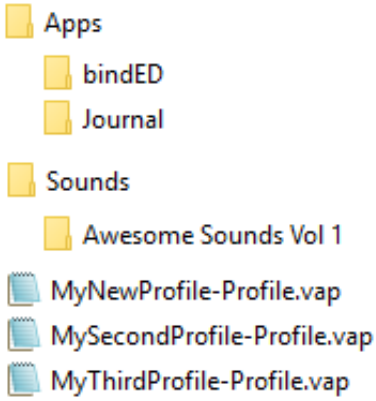
Supporting Tokens

You will be able to access various details about your command execution queues by using tokens. Information such as queue status (whether or not a queue exists, if it is running or paused), how many commands a queue has in it and whether or not a command is executing. See the section labeled, ‘Token reference’ earlier in this document for more info on these items (Note that the queue tokens all start with ‘{QUEUE’).

VoiceAttack Profile Package Reference

You can import a collection of files into VoiceAttack using a VoiceAttack package file. A package file can contain any number of exported VoiceAttack profiles (.vap), sounds or plugin/apps. A VoiceAttack package file is a **.zip** containing all the necessary files to make up a profile package, and has a '.**vax**' extension.
To import a VoiceAttack profile package, simply click on, 'Import Profile' from the Main screen. Select a package file (.vax) and the import will start. The files are copied into the folders you have specified for Sounds and Apps (on the Options screen), and the included profile files (.vap) are imported as profiles.
To create a VoiceAttack package to share, you simply zip together the necessary exported profiles (.vap) and the folders (sounds and apps) that you would like to include. The profiles MUST be in the root of the .zip (not inside a folder inside the .zip). Any sounds must be in a folder named, 'Sounds' in the .zip (the, 'Sounds' folder MUST be in the root of the .zip (not inside a folder inside the .zip)). The file structure *inside* the, 'Sounds' folder is what will be copied to the folder designated as the Sounds folder for VoiceAttack (what is set on the Options screen). Note that any file with the same path and filename will be overwritten (useful for large updates).
In exactly the same fashion as the, 'Sounds' folder, any plugins/apps that are in the package must be in a folder named, 'Apps' in the .zip (the, 'Apps' folder MUST be in the root of the .zip (not inside a folder inside the .zip)). The file structure *inside* the, 'Apps' folder is what will be copied to the folder designated as the Apps folder for VoiceAttack (what is set on the Options screen). Note that any file with the same path and filename will be overwritten (again, useful for updates). Note that since plugins are .dll files, the **plugin feature must be disabled** (and VoiceAttack restarted) prior to importing the, 'Apps' folder. Note that if you do NOT have any plugins/apps, do not include an, 'Apps' folder (there is a check for the presence of an, 'Apps' folder in the package).

Here is an image of a sample package that contains profiles, sounds and plugins:



Note again that everything is in the root of the zip (and not in a folder in the .zip lol).

Two additional, ‘hidden’ features of the package import are two flags that are included with exported profiles (version v1.6.5+). You’ll have to edit the .vap files with a text editor to get to them, though. The first flag is, ‘IO’ (for import order). You’ll see this flag somewhere near the bottom of the file as ‘<IO>0</IO>’. The profiles included will be ordered by this flag. So, if Profile, ‘A’ has an IO flag of ‘0’, and Profile, ‘B’ has an IO flag set to ‘1’, Profile A will be

imported first, followed by Profile B. Is order important? Probably not at the moment, but if necessary at some point, it's there. The second flag is, 'IS' (for import select). If this flag is set to 1 (or non-zero), VoiceAttack will switch to this profile when it is imported (just like when you import a single profile (.vap)). If this value remains zero, VoiceAttack will not switch away from the currently-selected profile when the package is finished importing.

The last thing to do is to make sure that your zipped up file is renamed with the extension of '.vax' so that VoiceAttack will pick it up. That's all there is to it! Since this is a relative advanced topic, you'll probably want to come visit everybody over at the VoiceAttack User Forums for more info: http://www.voiceattack.com/forum

Troubleshooting Guide

This section is going to be updated frequently, so, check back. Also, if this guide does not help you, please visit the VoiceAttack forums at **http://www.voiceattack.com/forum**

(edit – this guide is only covering some basic items which may help you in your particular situation. The following thread on the forum is going to help you the most with tough speech engine stuff (fairly new, fairly complete):

http://voiceattack.com/SMF/index.php?topic=1635.0 note that there are links in that thread to other threads that will help as well.).

Almost every single issue with VoiceAttack will have to do with VoiceAttack not hearing or understanding your commands.

Here are some things I've found while using VoiceAttack over the last year or so...

VoiceAttack not listening at all (Level bar on the main screen is not moving):

1) Make sure your microphone is plugged in all the way (this gets me just about every time).
2) Make sure your microphone switch is on. This is sometimes located on the cord of your headset.
3) Make sure the volume of your microphone is adequate. Also, make sure that it is not muted in Windows.

   Additional places to look:

   If you are using Microsoft Windows Vista/7/8/10, go into Control Panel & run the

   'Sound' app.

   Select the recording tab. Select your input device (will have a green checkmark next to it).

   Go to the levels tab. Make sure the volume is adequate (mine is up all the way). If there is a balance button, click it. Make sure that the microphone channels are at adequate levels (mine are both up all the way).

   In XP, go to the 'Sounds and Audio Device Properties' app. Click the 'Audio' tab & make sure the proper device is selected in the drop down list. Click on the 'Volume' button to make adjustments to the volume.

   Even more:

   If you are using Microsoft Windows Vista / 7 / 8 / 10, go into Control Panel and run the 'Speech Recognition Options' app.

   Select the 'Set up microphone' link. Follow the instructions indicated to set up your

microphone.

Make sure to check out the, ‘Setting up Microphone Input’ later in this document.

VoiceAttack is not understanding commands (Level bar is moving, but, VoiceAttack is not recognizing what I say):

1) Make sure your microphone levels are adequate as outlined above.
2) Make sure that the speech recognition engine is trained to your voice. This sounds a bit tedious, but, it will make a world of difference in how well VoiceAttack can understand you & will add levels of fun to your VoiceAttack experience. Seriously... I can't stress this enough. Go into Control Panel and run the 'Speech Recognition Options' app. Click on the 'Train your computer to better understand you' link. I did it three times. I don't know if doing it that many times was overkill, but, it certainly made things work really well.
3) Check to see if VoiceAttack is 'Listening'. It's a big button on the main screen on the right side. If it says, 'Not Listening', you need to click it :)
4) Make sure VoiceAttack is not already running a long or huge macro. You will see indication of this in the recognition log on the main screen (big, white list looking thing). Click the 'Stop Commands' button to halt your macro if you need to.
5) Are you using the right profile? Sometimes I'll switch to a different profile and forget that I did. Switch back over to the right profile :) (Note : this is for non-trial version only. There is only one profile in the trial version.)
6) Is your environment fairly quiet? Sometimes your chatty or noisy house mates will confuse VoiceAttack.
7) Take a look at the recognition log on the main screen. What does VoiceAttack 'think' you are saying? You might have to make adjustments to your command (given that you have trained up the speech recognition engine as outlined above). Just a note... we have thick accents down here in Texas. Sometimes what I am saying is not recognizable even by other humans :)
8) Out on a limb... make sure you didn't change to a speech engine that you did not train up (you'll find this on the options page). Try switching to 'System Default'.
9) Even more out on a limb... Make sure you are not running any kind of voice-altering software (you know the kinds that make you sound like an alien or an orc or whatever). Depending on your software, these effects may be passed straight into VoiceAttack, which, in turn, will probably not recognize what is being said.

VoiceAttack is recognizing what I am saying (commands are showing up in the log), but nothing is happening in my game.

Occasionally, you may have to adjust your commands in VoiceAttack to work with certain games. Here are some things you will want to try:

The **most** common issue is that VoiceAttack is sending key presses too quickly to your game.

If a game is polling for input at a specific time, and, VoiceAttack's key press does not coincide with that polling, the game will simply not catch it. To fix this, you need to increase the amount of time a key is held down before releasing (see, 'Key Press Screen' – 'Hold down for X seconds' option). A good place to start is 0.10 seconds. Try increasing and decreasing this number to see what works best.

Another common, frustrating issue is that some games or apps will require VoiceAttack to be run in Administrator mode to allow interaction. To run VoiceAttack in Administrator mode, do the following:

Locate the file called, 'VoiceAttack.exe' (usually in C:\Program Files (x86)\\VoiceAttack folder). Right-click on the file and select, 'Properties'. Go to the, 'Compatibility' tab and then check the, 'Run as Administrator' box. Click, 'OK' and then run VoiceAttack. Another (maybe easier) way to run VoiceAttack as an administrator is to go into the Options screen and then open up the System/Advanced tab. On that tab you'll see, 'Run VoiceAttack as an Administrator'. Check that box and VoiceAttack will attempt to run as an administrator the next time it starts.

One last thing to check is to make sure VoiceAttack is sending input to the right place. Verify that your commands are being sent to the proper process, or, to the Active Window (see, 'Voice Attack's Main Screen' – 'Target' option).

As always, if none of these methods work, please check out the VoiceAttack user forums :)

Setting up Microphone Input

Way down here, gleaned from the VoiceAttack User Forums, is a short guide to hopefully help you set up your microphone input for use with VoiceAttack and Windows speech recognition. It just goes over the basic steps that we sometimes miss when setting things up.

Audio input and output in Windows is somewhat of a maze, but makes sense after you learn your way around. As with everything, you set something once and then forget how it's done. This is here for both you and me :)

Firstly, VoiceAttack uses Windows' built-in speech recognition to do its thing. Out of the box, the speech engine uses the default recording device to receive input. The recording devices can be accessed by right-clicking on the, 'Speakers' icon in the system tray and selecting, 'Recording Devices', or, by right clicking on the VoiceAttack icon in the task bar and selecting, ‘Recording Devices’. Right-clicking on the desired microphone/headset and selecting, 'Set as default device' will do just that. You will know your device is selected as default by the green check mark appearing over the device's icon. This works for most, since we've usually only got one device on our machines that we primarily use.

For those that have multiple microphones/headsets/webcams, the default device is sometimes not what you want selected. To change the recording device that the speech engine uses, just open up Control Panel and choose the Speech Recognition applet, or, right-click on the VoiceAttack icon in the task bar and select, ‘Speech Control Panel’. Click on the, 'Advanced speech options' link and then click on the button at the bottom labeled, 'Advanced...'. This opens up the, 'Audio Input Settings' dialog. There are two choices in this dialog: 'Use preferred audio input device' and 'Use this audio input device'. The first choice allows you to continue to use the default input device (whatever that device might be). Choosing, 'Use this audio device' lets you choose a specific device from the dropdown (default recording device is ignored).

VoiceAttack’s Data Storage

VoiceAttack stores its data in a single file named, 'VoiceAttack.dat'. Each user on your PC will have their own VoiceAttack.dat file that will be stored (usually) in "C:\Users\YOUR_USER_NAME\AppData\Roaming\VoiceAttack". If you want to back up your VoiceAttack data in one big chunk, the VoiceAttack.dat file is the one you will want to save. Accessing this folder for the first time may be tricky, as Windows may have these folders hidden. What you will want to do (if you haven't already done so) is to have Windows show hidden files and folders: https://support.microsoft.com/en-us/help/14201/windows-show-hidden-files Another (probably quicker) way to browse this folder is by clicking on the link provided on the Options screen, under the, ‘System/Advanced’ tab labeled, ‘Click here to browse VoiceAttack’s data folder’.

On a side note, and to clear up some confusion, VoiceAttack's data is NOT stored in .VAP files. .VAP files are the files that are created if you decide to export a single VoiceAttack profile (to share or simply as a backup). You may have had to import a .VAP file at one time or another. The information contained in the .VAP file that you imported is merged into your VoiceAttack.dat file and saved.

VoiceAttack Author Flags

There may be times when you need a finer level of control over certain aspects of your profiles when it comes to releasing them out into the community. Below are a few flags that can be altered within commands and profiles to help shape the delivery of the content you create. Currently, there is no user interface to edit these items. You will first need to export your profiles as a standard .VAP that is not compressed (see, 'Exporting Profiles' earlier in this document). You must then use a text editor (like Notepad) to modify the XML data, save the .VAP and then re-import the profile back into VoiceAttack.

Note: If the intent of using the supplied author flags is for obfuscation purposes, please be advised that VoiceAttack.com does not accept any responsibility for the use of these flags, as the use of any/all of the flags is solely up to the author to employ them. As we are all well aware, any attempt to obfuscate any type of data is an open invitation for anybody to defeat that obfuscation, so no guarantee is made that your profiles and commands will be completely hidden.

Profile-level flags

Inside the XML of the .VAP, within the Profile element, you will find a few elements that can only be modified outside of VoiceAttack. In the section above titled, 'VoiceAttack Profile Package Reference', we went over the <IO> and <IS> elements when creating a profile package (you can find out more about those in that section). In addition to those flags, the Profile element also contains a few more:

<IP> - The IP element will let you indicate to your users how you wish to manage the importing of your individual commands into other profiles. This is handy if you have commands in the profile you designed that are dependent on each other and allowing the commands to be imported individually into other profiles would cause those commands to not work (and, basically create a bottomless support pit). You can specify that a warning is displayed, or, a full stop message. To display a warning to indicate that individual command imports may not work on their own, and that you recommend importing the entire profile, change the IP element to '1': **<IP>1</IP>**. To prevent individual commands from being imported into other profiles at all, change the IP element to, '2': **<IP>2</IP>**. To disable, simply change or leave the value as, <IP>0</IP>. Probably not something you will use a lot, but it's there if you need it.

<IPM> - The IPM element will let you specify a message to display to the end-user if you've specified an <IP> tag above. If you do not specify a message in the <IPM> element, "The author of this profile has indicated that directly importing commands may cause issues and recommends that the profile be imported as a whole in order for the commands to work properly. Please see the profile author's documentation for more details. Do you still wish to continue?" will be displayed if <IP> is set to 1, and "The author of this profile has indicated that the commands cannot be imported directly, and you will need to import the profile in its entirety. Please see the profile author's documentation for more details" if <IP> is set to 2.
<IPM>Commands may not be imported directly into this profile, silly goose.</IPM>

<BE> - The BE element indicates that the profile can **only** be exported from VoiceAttack as compressed binary or an HTML command list (and not editable XML). To turn this flag on, simply set the BE element to, '1': **<BE>1</BE>**. To disable, simply change or leave the value as, <BE>0</BE>. **Note that if you distribute your profile initially as a compressed binary and this flag is set to 1, there is no provided way to change that profile back.** So, if you plan on distributing a compressed binary with this flag set, you will need to maintain at least two versions of your profile. A version that is used for authoring, and a version that will be used for distribution.

<AuthorTag1>, <AuthorTag2>, <AuthorTag3> - The AuthorTag elements will allow you to indicate whatever you would like as text values that persist through export. These values can be retrieved by calling the proxy methods, 'Profile.AuthorTag1()', 'Profile.AuthorTag2()' and 'Profile.AuthorTag3()'. See, 'VoiceAttack Plugins (for the truly mad)' earlier in this document.

<InternalID> - The InternalID element is a GUID flag for use by authors to be able to identify a profile. This value can be retrieved by calling the proxy method, 'Profile.InternalID()'. See, 'VoiceAttack Plugins (for the truly mad)' earlier in this document.

<AuthorID> - The AuthorID element is a GUID flag for use by authors to indicate the creator of the profile. This value can be retrieved by calling the proxy method, 'Profile.AuthorID()'. See, 'VoiceAttack Plugins (for the truly mad)' earlier in this document.

<ProductID> - The ProductID element is a GUID flag for your use by authors to distinctly identify profiles they create. This value can be retrieved by calling the proxy method, 'Profile.ProductID()'. See, 'VoiceAttack Plugins (for the truly mad)' earlier in this document.

<CR> - The CR element will allow you to restrict the end user from adding new commands to a profile (this also includes importing commands). This is primarily for protection against data loss in the event of profiles being overwritten, and encourage end users to, 'include' the profile rather than modifying it directly. Simply set this element to 1 to turn it on: **<CR>1</CR>**.

<CRM> - The CRM element works in conjunction with the <CR> element above. This will allow you to specify your own message about why commands are not allowed to be added. For instance, you may want to protect the end user from adding commands to prevent data loss due to a profile overwrite. The default message is, 'The author of this profile has indicated that new commands are prohibited.'. **<CRM>Commands may not be added to this profile, as this profile should only be, 'included' and never modified. </CRM>**

<CLM> - The CLM element at the profile level serves as a blanket message in case you do not want the overhead of indicating a CLM element for every command that is locked. Note that you can override this value at the command level (see <CLM> in the Command-level

flags section below). **<CLM>Commands in this profile are locked for your own protection. Carry on.</CLM>**

<PD> - The PD element will allow you to restrict the end user from duplicating your profile. Simply set this element to 1 to turn it on: **<PD>1</PD>**.

<PDM> - The PDM element will allow you to provide a message to display to the end user if you've specified the <PD> flag above. If a message is not provided in the <PDM> element, "The author of this profile has indicated that profile duplication is prohibited" will be displayed to the user. **<PDM>Profile duplication is prohibited, silly goose.</PDM>**

<OP> - The OP element will allow you to indicate that you want to overwrite an existing profile when the profile is imported. The existing profile must have an InternalID that matches the InternalID of the profile being imported. Set this element to 1 to turn it on: **<OP>1</OP>**.

When the OP element is used, the CO and CV elements can be used to indicate the following:

<CO> - The CO element allows you to copy and preserve the profile options from the profile being overwritten into the profile being imported (using the OP element). Set this to element to 1 to turn it on: **<CO>1</CO>**.

<CV> - The CV element allows you to copy and preserve the profile's saved variables from the profile being overwritten into the profile being imported (using the OP element). Set this to element to 1 to turn it on: **<CV>1</CV>**.

<PE> - The PE element will allow you to restrict the end user from exporting your profile. When this option is turned on, the profile can only be exported as 'HTML'. Simply set this element to 1 to turn it on: **<PE>1</PE>**.

<PR> - The PR element is for future use.

Command-level flags

Below are the available command-level flags that you can use. Again, you'll only be able to access this flag by editing an uncompressed .VAP file with a text editor.

<InternalID> - The InternalID element will allow you to specify your own GUID value to identify an individual command. This value can be accessed from the proxy method Command.InternalID(). See, 'VoiceAttack Plugins (for the truly mad)' earlier in this document.

<CL> - The CL element will allow you to lock an individual command's contents from being accessed on its own (edited, viewed, duplicated, copy/pasted, copied to other profiles or

imported into other profiles). This is handy for commands that are complex and allowing access would probably open up support issues, or, maybe you just don't want to share your work with everyone. The type of command you would use with this flag would most likely be a subcommand, as your users would not be able to edit any aspect of it (spoken phrase, hotkeys, category, description, etc.). Set this element to '1' to turn it on: **<CL>1</CL>**.

<CLM> - The CLM element works in conjunction with the <CL> element above. This will allow you to specify your own message about why the command is locked. For instance, you may want to protect the end user from changing details about the command due to an impact on performance or loss of functionality. The default message is, 'The author of this profile has indicated that the content of the selected command is locked and cannot be accessed.'. Note that this message overrides the CLM element indicated at the profile level (see above).
<CLM>This command is locked for your own protection. Carry on.</CLM>

Note that if your profiles are distributed uncompressed, or, compressed and not using the <BE> flag (above) set to, '1', the end-user will still be able to modify these flags, so, plan accordingly.

For fun... maybe

Welcome to the end of the help document. Thanks for reading my mishmash that´s been growing for several years :) Some extra things thrown in, just for fun (or not) will be down here as I add them (or recall that I added them and neglected to document).

If you right-click on the VoiceAttack icon in the top-left corner of the main screen, you'll see an option for, '**Cover of Darkness**'. Checking this option this puts the main screen in, 'dark mode', which might be useful when using VoiceAttack at night. To go back, just click on the menu item.

Command line parameter, '**-opacity**' was added as a test. Passing a value of 0 to 100 affects the main screen's opacity level. 100 = not transparent at all, 0 = fully transparent. Example: -opacity 75 sets the opacity at 75%.

To override the default on/off sounds in VoiceAttack (listening on/off, joysticks on/off, etc.), just add a valid .wav file called, '**sys_on.wav**' and/or a file called, '**sys_off.wav**' to the same directory that VoiceAttack.exe is located (usually C:\Program Files (x86)\VoiceAttack). If you want to override the default interrupt sound (stop all commands.), just add a file called, '**sys_stop.wav**'. Note that these must be valid .wav files. If an error is encountered, the sounds are reverted back to the default sounds. Note also that these sounds will override any selected sounds on the options screen.

If you want to back up your VoiceAttack.dat file (the file that holds ALL your profiles), it is usually located in C:\Users\YOUR_USER_NAME\AppData\Roaming\VoiceAttack.

The, 'Backup' directory located in C:\Users\YOUR_USER_NAME\AppData\Roaming\VoiceAttack holds up to the last ten changes made to your VoiceAttack profiles. To roll back to a previous change, simply move any one of the files out of the Backup directory up into the C:\Users\YOUR_USER_NAME\AppData\Roaming\VoiceAttack directory and replace the current VoiceAttack.dat. You'll probably never ever use this, but it's there if you need it ;)