

Dragon Sample Commands Commentary

NOTE: This is draft material, further updates will be posted late-February 2007.

Nuance includes several advanced sample commands with Dragon NaturallySpeaking®. These are normally installed in the directory:

```
C:\Documents and Settings\All Users\Application Data  
  \Nuance\NaturallySpeaking9\Data\Enx\samplecommands
```

Upon installation, a shortcut to this directory is placed in the Dragon NaturallySpeaking 9 program group so that you can go to Start, All Programs, Dragon NaturallySpeaking 9.0, and then MyCommands Samples.

The commands are in XML format. There is a readme.txt file and an associated Word template for one of the command sets. Seven sets of commands are included. When imported, the commands go into the “Dragon Sample” command group, so that once finished with the commands, they can be easily deleted from your system from the Manage tab. Like most of the samples in this book, these are not intended to be used as-is, but are intended to be modified as needed.

These commands focus on the use of the features introduced in Dragon NaturallySpeaking® 8 providing “Structured States” for command recognition. This is an expansion of the concepts of having application-specific commands and window-specific commands.

One of the samples provided, the Endoscopic Biopsy sample, does not deal directly with Structured States. It provides an example of how to do similar tasks without using states, using only the new “PromptValue” method.

The following commentary presumes that you have imported these commands into a Release 9 Dragon NaturallySpeaking®. This is done by opening the Command Browser, going to the Manage tab, selecting Import, navigating to the above directory, changing the “Files of Type” to show XML files, and then importing the command sets, one at a time. All of these illustrate relatively advanced concepts and techniques for creating commands.

Following is additional commentary on these commands. It is anticipated that the reader will read the readme.txt file and the comments within the text of these commands. The order in which these commands are discussed is different from that presented in the readme.txt file associated with these commands, instead starting with the simplest command and progressing through the most complex.

When reviewing these commands, use the Dragon option to have DragonBar at the top of your screen even if you normally have it in Tray Icon or some other mode. Otherwise you will not see some of the command behavior.

These commands are not intended as illustrations for casual users. They are illustrations of command technique for persons willing and able to design moderately-complex sets of commands for particular applications. The commands themselves use a wider variety of VisualBasic and VisualBasic for Applications features than the prior examples.

SetStateSample

The text of this command is found by going to the Script tab of the Command Browser, setting the Current Command Grammar to “Dragon NaturallySpeaking – DragonPad,” highlighting the “*sample set state*” command, and selecting either Edit or New Copy.

This is the simplest of the command sets dealing with states, and illustrates the basic concept. Understand it before moving on to more complex groups of commands.

The key concept is simple: set a “State” and then the commands defined as being recognized in that “State” should be recognized.

This is an artificial command built only to show the concept. Don’t look for any hidden meaning in what this command does.

After giving the command “*sample set state*” then give one of the directly-related commands defined in the <greetings> command. These commands are:

Good afternoon

Good day

Good morning

Good evening

After doing so and getting the red font response message, then give a global command such as “*show dictation box.*” Seeing that command work normally should remind you that the “State” does not exclude global commands. Dictate plain text and you will note that being in a “State” does not exclude dictation. If you attempt an application-specific command, it should also work. By itself, the SetState adds commands that are defined in the appropriate State...in this case, the “RedFontParrot” State.

The State itself is invisible.

The “*goodbye*” command references the GetState function in a simple fashion to determine whether the “RedFontParrot” State is active or not.

Near the end of the “*goodbye*” command the ShowMessage method shows how to place text on the DragonBar. This method is useful in applications where simple instruction or prompting is needed and the DragonBar is known to be present. Such messages do not show if the DragonBar is not present as when the Tray Icon mode is selected.

(As an exercise for the reader, modify one of the following commands to display the State in the DragonBar using ShowMessage as the State changes. It might assist you in following the command logic.)

After giving the “*goodbye*” command, try the “*good afternoon*” command again. It should not be recognized as a command as the State is no longer the RedFontParrot State.

Grocery List Sample – DragonPad

There is a set of 4 commands comprising the GroceryListSample_DragonPad. In addition to the techniques introduced in the “sample set state” command it introduces the PromptValue function.

The commands are:

prepare grocery list which must be the first command given

<producelist>

<dairylist>

<meatlist>

The commands:

<producelist>

<meatlist>

<dairylist>

are all almost the same. Each of these recognizes a set of food names plus the key phrases “*what’s available*” or “*next*” as commands. There are slight differences because <...> is set up to be the first part of the grocery list, <...> the second part, and <...> the last part.

In the Manage and Script displays, notice the Current Structured State box for selecting the State for display.

Within the Browse display, there is no evidence of Structured States. In the Browse display, the structured state commands appear, possibly leading to confusion about valid commands within particular applications. That is, a command appearing in the Browse display might be a valid command only within a specific State within that application.

“*Prepare grocery list*” is the initial command – consider it to be the opening of a form. It progresses through the glProduce State, then glMeat, then glDairy before returning to the null (“”) State. By controlling the State, only one group of food items is available at a time to be recognized as commands.

At the start, when the State has not been defined and is implicitly null (“”), then the “prepare grocery list” is the only command out of this set recognized.

Once this command is entered, it soon changes the State to “glProduce.” That ADDS the <producelist> commands to those available to be recognized. Note that other DragonPad and global commands are available, as is general dictation. This has only ADDED to the list of recognized commands. That is, while in the <producelist> state, you can dictate “steak and eggs” and it will be recognized and added to the list as dictation. Remember this is only a sample of a technique!

If command mode is set, then dictation will not be active and the only commands available may be those in a specific state. That is appropriate in many applications where a single field has few values.

But the <dairylist> and <meatlist> commands are NOT recognized as commands. If you dictate any command from those lists, they will be recognized as text.

There are several sets of code deserving close attention in the “*prepare grocery list*” command. These include a set of code for formatting within DragonPad, WordPad, or other RTF editors, error handling using the On Error statement, the ShowMessage code for presenting text in the DragonBar, and the PromptValue code for presenting a prompt with a limited response set.

The statement:

```
ExecuteScript "SetCharFormat "bold=1,underline=1"" ,1
```

is one way to force font bold and underlined. It works within DragonPad, WordPad, or other "RichEdit" controls regardless of the current state of the font. In theory the ",1" allows the script to finish. However, sometimes such scripts do not actually complete before the next line of the command is executed.

The "SetCharFormat" is part of the Legacy scripting and is not documented in the manual or the help files for Dragon NaturallySpeaking® 9. It is documented in the manuals for Dragon NaturallySpeaking® 5 and earlier releases. While it may require a wait state, it is a handy way of controlling formatting in applications like DragonPad and WordPad. In Word or WordPerfect there are much better means of controlling format.

In testing this command, several times I observed headings in mixed formats, as if the bold and underline was applied while the next command was processed. For example, the heading:

MEAT:

was seen. To avoid such problems when using this technique, add a short Wait statement after the ExecuteScript command.

The SendKeys "^e",True is to center the heading. "Ctrl+e" are the keystrokes to center a line. SendKeys "{Enter}^1{Enter}",True left-justifies the text. "Ctrl+l" is the left-justify keystroke sequence.

If interested in performance, the next two Wait statements could be eliminated by combining the statements:

```
SendKeys "^e", True
SendKeys "GROCERY LIST", True
Wait .2
SendKeys "{Enter}^1{Enter}", True
Wait .2
SendKeys "PRODUCE:{Enter}", True
```

into one SendKeys statement.

```
SendKeys "^eGROCERY LIST{Enter}^1{Enter}PRODUCE:{Enter}"
```

The next ExecuteScript command sets the format back to normal.

The On Error Resume Next illustrates one method of error handling. While it might seem that the next statement could never fail, Dragon NaturallySpeaking® generates an error if it finds no commands in State named in the SetState statement.

The SetState = "glproduce" sets the state. The CheckErr function is called to determine if there is an error.

Within the CheckErr functions, provision is made to display an error. For longer commands where CheckErr might be called multiple times, the "CallersLine(0)" function might be added so that you could get the line number of the CheckErr statement.

Next, the long statement:

```
EngineControl.DragonBar.ShowMessage 0, _  
"Say ""What's available"" to see the list of available items"
```

puts the string into the DragonBar. You may see a truncated version of this message on your DragonBar depending upon your system settings. The ShowMessage function is handy for giving a user hints of what are acceptable responses, displaying error conditions, limitations, etc. The effective length of the text that is actually seen in the DragonBar is dependent on system settings including fonts so should be kept short. Some of the sample messages are too long to fit on the DragonBar.

At this point, the *"prepare grocery list"* command is finished but the task is not complete.

The commands in <producelist> are now recognized. Edit that script to view the complete text of the <producelist> command.

What may not be obvious is that the array Values is used to define the commands that will now be recognized. These do not have to be commands in the <producelist>! They can be global commands or they can be application-specific commands. The commands named in the array Values, which is referenced in the statement:

```
Item = EngineControl.PromptValue(Values(), "Available Produce"
```

Will be available to be recognized.

It appears that Dragon NaturallySpeaking® may give some priority at this point to the commands in the PromptValue array, but documentation is murky on this point.

You may test this by setting one of the Values to "" instead of the vegetable name. Save the command, then try dictating the vegetable name that you effectively deleted. If you are in command mode, it won't get recognized. If in dictation mode, the word you say may still be recognized as dictation.

GroceryList Sample – WordPad

This set of sample commands uses the same concepts and coding structures as the DragonPad examples.

TGV Restaurant Sample

This sample shows how Text and Graphics commands can be blended together with Advanced Scripting commands. It also illustrates how exactly the same command can be set up to present different values in different states.

To start this sample, say "*Chinese Menu*", "*Japanese Menu*", or "*Sushi Menu*" (do not follow the instructions in the readme.txt file). Where the prompt says "...Say 'item number' and a number to order by number" ignore this and just say "*item*" followed by a number as in "*item 3*".

It uses the same SetState, GetState, and EngineControl.DragonBar.ShowMessage features that the other commands use.

For practice changing commands, as an exercise for the reader, modify the commands to conform to the instructions provided. This requires at least three changes to command names.

Colon Cancer Checklist

This command operates in Microsoft Word. If not using Word-2003, follow the instructions for changing references before attempting the command.

Before looking at the scripts, use the command first in Normal Mode, then switch to Command Mode and execute the same script.

After giving the "*colon cancer checklist*" command, give the command "*choose from list*" to see the full impact of this command. Then follow the prompts. When in Normal Mode you should notice that you may continue to dictate free text into these fields, or else dictate what is given in the prompts. When in Command Mode the choices are much more restricted.

Like the prior examples, this uses SetState, GetState, and PromptValue. No new Dragon concepts are presented, but within this command are examples of how to use some Word constructs along with Dragon.

Endoscopic Biopsy

The complete name of this command, "*<form_action> endoscopic biopsy gross template*" is rather long. Practice changing command names by giving this command a shorter name.

This sample command illustrates how one command can be used to completely fill out a substantial report. To complete it by voice requires that one be in Normal Mode to allow free dictation into the last field. Unlike most of the other sample commands, it does not use the SetState or GetState to modify the available commands.

The "commands" that are used to generate the text are actually prompt values in this example. This illustrates a concept that may have a wide range of uses. These prompt values essentially have a written and a spoken form. The "written form" is what is returned as the result of the EngineControl.PromptValue. While this example displays the "written form" one can utilize that value to do other actions within the script.

This example illustrates the use of a null value (empty string) to form a separator. Notice how the prompts have a line after the "next" and "normal" entries. That line is a direct result of the setting of Values(3) to "". To further illustrate this, change the line (arbitrarily chosen):

```
Values(7)="brown\brown"
```

to:

```
Values(7)=""
```

And repeat the commands. Another horizontal separator within the prompts for color will be seen.

The prompt values need not be coded directly in the code as illustrated. They may be read from a file, obtained from a database, or constructed from Outlook data.

Sample Injury Report

The Sample Injury report is an example of how to use Dragon NaturallySpeaking® within a Word Form. It hints at mechanisms to use Dragon NaturallySpeaking® interfaces within other Dragon NaturallySpeaking® macros.

The entire macro set for this example is several pages long. For those not familiar with Word forms, there is processing that may not make sense. But for those who use forms, there are several good ideas embedded in this sample. Please refer to the macro set as delivered, as there are several comments not repeated.

The Word macros are several pages long. They illustrate how one can introduce commands specific to a field within the context of a Word form.

A link to an annotation of the Word macro is posted at the web site:

<http://www.pcspeak.com/books/>

It is advanced material, much more advanced than the other material discussed in this chapter. It is suitable for programmers, persons who have implemented Word Forms, and others with a strong interest in automating data entry.