A.K.A. – Using PowerShell Aliases

An interesting capability baked into the PowerShell environment is alias commands. Much like we can find in IOS (The Cisco one that is) and with other CLI environments, there are short commands available to save you some typing during your day-to-day scripting.

Aliases can be found very easily with the Get-Alias CmdLet. The resulting output of the **Get-Alias** CmdLet is a table of aliases and their associated full CmdLet name.

There are a few different opinions about using aliases versus using full command syntax. My preference for writing scripts is to use full CmdLet structure. This is for readability, and because quite often I'm sharing my code with my peers who may have less familiarity with the commands. With that said, I do use aliases in the interactive shell to save time.

Aliases that conflict with commands

This is something that you may discover if you were used to using the command line for administration with other tools like the Windows Resource Kit, or even with native command line tools that came with Microsoft Windows.

An example of this is the **SC.EXE** tool. This is the Service Control utility which was a Resource Kit tool with Windows 2000, but was packaged into the OS in recent releases. You can use this to start, stop, install, uninstall and perform other management tasks with Windows services.

The challenge with this comes when you are using the PowerShell command shell there is a conflict. In PowerShell SC is an alias for the **Set-Content** CmdLet.

Traditional DOS command shell

In a regular DOS shell (cmd.exe) we run the SC command and get the expected results because it locates the SC.EXE in the system path as we see here:

PowerShell command shell

In the PowerShell session we get a different result because the priority to PowerShell CmdLets and functions trumps the Windows path and this is the result when we try to use the **SC** command:

This is one example, but there are others that you may encounter depending on what tools you use. The workaround is to test your scripts thoroughly, and where conflicts occur, you should use the **Invoke-Expression** CmdLet referring to the literal path to the executable file.

Setting your own Aliases

One of the creative ways to leverage PowerShell is to create your own aliases for commands and functions. For those who are new to PowerShell, this may not be the first thing that you run to, but once you begin to expand your PowerShell toolkit to include your own functions you will quickly find that aliases can add that extra layer of awesome to your shell.

As an example, I want to create a function named DiscoPosse which will search the current directory for a file which contains the phrase DiscoPosse in the file name.

```
function DiscoPosse { dir . | where { $_.name -match “DiscoPosse” } }
```

And once we define our function, we can call it by typing the function name in the shell as seen below:

Another cool feature is that once we define a new function you can use the auto-complete by typing the first part of the command (e.g. discop) and then hitting the Tab key to complete the command.

Let’s say that I want to assign the alias diggity to my new DiscoPosse function. The way to do this is to use the **Set-Alias** CmdLet with the syntax of ***Set-Alias <alias name> <command name>*** as follows:

```
Set-Alias diggity DiscoPosse
```

Now we test out our alias by typing diggity in the PowerShell command line to see the results:

Pretty cool isn’t it? Once again we can also use the Tab auto-complete capability because it also extends into aliases.

I hope that this is a useful tip, and in a future post I’ll delve further into how you can back up and restore your custom aliases to allow you to transfer them from machine to machine. In the spirit of “less is more” you will learn to love aliases as a part of the great PowerShell toolkit.