ControlUp Scripting Standards

Version 0.2, October 2019

Guy Leech

Ton de Vreede

About the Authors

Guy Leech leverages over two decades of hands-on End User Computing technical expertise to provide consultancy services for Citrix, VMware, and Microsoft ranging from upgrades, troubleshooting and optimization to whole infrastructure design and implementations. He is a current Citrix Technology Advocate (CTA) and former VMware vExpert.

Ton de Vreede consults as an architect, engineer and project leader on (migration) projects from traditional client/server environments to server-based computing environments. His experience also includes script writing and editing (using PowerShell, NT shell, WSH, VBA, KIX and Altiris, amongst others) for maintenance and configuration.

Introduction

With a number of people employed by ControlUp (CU) to write Script Actions (formerly Script Based Actions (SBAs)) and potentially a large number of external contributors, ControlUp is defining corporate standards to adhere to when producing scripts. The reasons are thus:

1. Make scripts more uniform in order to allow others to read, maintain and enhance scripts other than the original author
2. Potentially improve code quality and thus reliability and reduce support cases
3. Ensure the scripts look and behave in a professional manner which benefits a company of ControlUp's standing

ControlUp does not want to stifle creativity and force people to code in a style that they are not comfortable with. We do want to provide scripts to customers that are consistent and follow industry best practices where appropriate.

The following sections define ControlUp script requirements, with sample scripts to serve as references as well as a template.
All scripts will be written in English although they should be capable of running on systems where the locale is not necessarily set to English. Consider localized elements, like date format and event log language, and keep them in the same format where possible. For instance, when outputting a date, always use the 'G' format specifier with -Format so that it is formatted to the current locale and do not use separators for positive numbers, for example 1000 and not 1,000.

Commenting

Adding relevant annotation is key to helping others understand or modify your script.

- ★ What is obvious to you may not be obvious to everybody
- ★ What is obvious to you now may not be obvious next time you look at the script
- ★ WHAT a piece of code does may be obvious, WHY may not be

The following must therefore be adhered to:

- Scripts must have a PowerShell help format comment block at the top as documented here so that the purpose of the script, examples, and any required and optional arguments are easy to see. In order to keep this section compatible with the PowerShell help system (for instance, if people use the script outside of ControlUp), then Get-Help works with it, we use a few custom keywords but if these are prefixed with a . then the Get-Help function will fail to display the help so they cannot adopt the same format. This is the framework to use:

```
<#
      .SYNOPSIS
            One line as to what the script does
      .DESCRIPTION
            More detail, especially any non-passive actions
      .PARAMETER parameter1
            A description for each parameter which can be passed
      .EXAMPLE
            One or more to illustrate use
       .CONTEXT
            What is the CU context(s) , e.g. Computer, Session,
     Process, Advanced
      .MODIFICATION_HISTORY
            Full name - When (date format DD/MM/YY) - What changed

Optional:

.LINK --> externals sources/help & link to any code used from other
sources/authors
.COMPONENT --> all external components/pre-reqs such as required
modules
.NOTES → Anything extra worthy of note (optional section)
#>
```

- There should be a reasonable amount of relevant comments in the script, particularly if a particular area was difficult to code/implement or some kind of workaround was required. When submitting a script to ControlUp, always assume that someone else is going to debug or change your script—so help them (and yourself) as much as you can by adding explanations in comments when explanations will help the script reviewer understand your approach and logic, especially if you tried an another method first and that was too slow or failed.
- Comments should be indented to the same level as the statements they relate to. As an alternative, comments can be on the end of a line as long as their addition doesn't make the line too long to read without scrolling with a "reasonable" font size on a full HD display.
- Ensure comments are updated/deleted if the code to which they apply changes so that the comment no longer applies.

General Script Settings

- The first line of the script must be a #requires line. Please avoid using PowerShell versions higher than 3.0 unless it makes the script much more difficult or impossible to code. Why? Some ControlUp customers do not update PowerShell and use 2.0 on Windows 7 and Server 2008R2. If you're not sure of the version number, you can check it in code via $PSVersionTable. This is an important step because if you use a PowerShell version that's different than what's on the #requires line, it may result in an obscure error message even if the non-2.0 code is not executed, which makes it difficult to troubleshoot.
- Immediately after the help comment block at the top of the script, add the settings for the preference variables which control error actions and verbose and debug messages. Adding those settings ensures that errors are not ignored and are dealt with by the script (more about errors later). It also ensures that no verbose or debug messages are output in the production version but can easily be switched on by editing the script or by adding arguments which when present/true will set the verbose and debug preferences to 'Continue' as with the "Analyze Logon Durations' (ALD) script:

```
$ErrorActionPreference = 'Stop'
$VerbosePreference = 'SilentlyContinue'
$DebugPreference = 'SilentlyContinue'
```

- When developing/testing scripts, always run with strict mode set to latest to aid in capturing errors, but avoid adding this in the script:

```
Set-StrictMode -Version Latest
```

Script Parameters

- Up to and including ControlUp 7.4, parameters are passed positionally rather than by name and optional parameters are omitted when passed to the script rather than passing NULL, an empty string, etc. In order to make the parameter passing easy to understand in the script, a standard PowerShell Param() block should be used where the parameters are specified in the order that they will be passed so there is no need to use the "Position" keyword inside a "Parameter" directive.

  For instance, the following parameters defined in an SBA thus:

#	Record type	Caption	Default	Input Validation
$args[0]	Computer	Name	N/A	
$args[1]		Param2		
$args[2]		Param3		

  Will be consumed by the corresponding SBA thus:

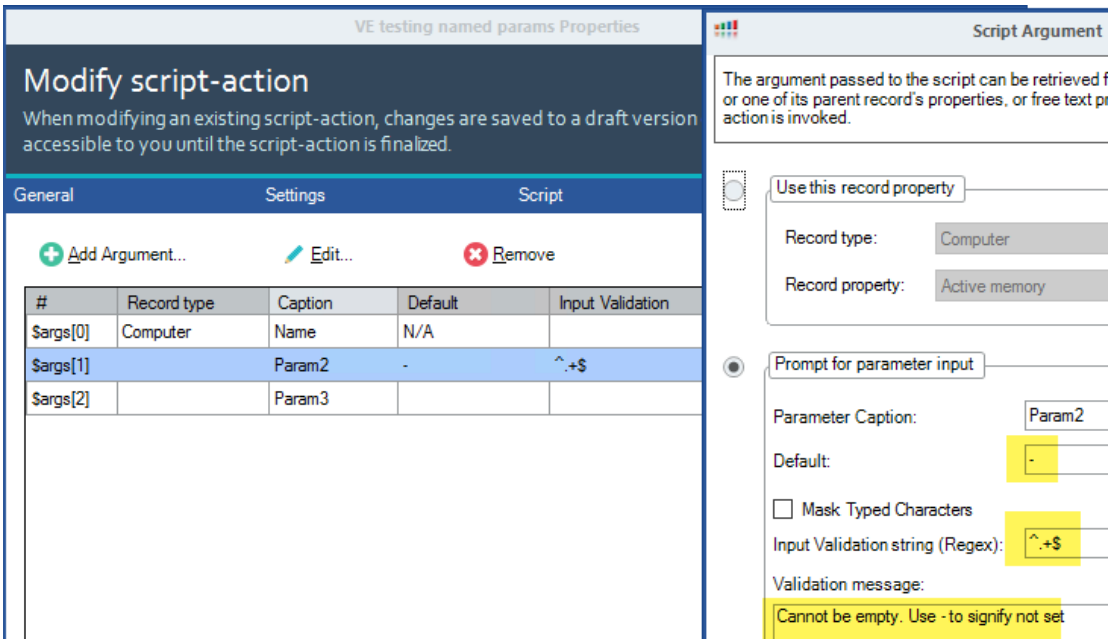
```
[CmdletBinding()]

Param
(
    [string]$computerName,
    [string]$parameter2 ,
    [string]$parameter3
)
```

Missing or optional parameters in the CU console should be placed last in the list when defining them in the console and then checked thus before using them in the script.

```
If( $PSBoundParameters[ 'parametername' ] )
```

Note that in the above example, if $computerName and $parameter2 are present and $parameter3 is not, the script will struggle to decide if $parameter3 is not present because a) it simply wasn't specified when the script was invoked or b) because $parameter2 wasn't specified so $parameter3 has taken its place.

For this reason, only one optional parameter is allowed, which ideally should be placed at the end of the arguments. If other arguments don't need to be specified then set a default of something like '-' and a regular expression that does not allow empty arguments and have the script treat '-' as a special case, setting the parameter to $NULL, -1 or 0, etc depending on the type of argument. For example:



- Use plain yet descriptive language. Avoid shortening anything to reduce the risk of typos. ISE and VScode tab completion can exacerbate typo-related problems (although using "Set-Strictmode -version Latest" will assist in finding them). As an example, use "$computerName" rather than "$strComputerName" but strongly type the definitions, i.e.:

```
[string]$computerName = $null
```

- Always specify the variable type, bearing in mind that some variable types don't exist in some classes (e.g. a .NET class), and the code can deal with it, e.g. outputting a meaningful error that you need to install some component/module.
- Declare the parameters with the ones most frequently used first and the ones least used last so when tabbing through parameters, the most frequently used appear first, making it slightly quicker to use the script outside of ControlUp.
- Always declare a short help string for the parameter to aid understanding of what it is, e.g.

```
Param
(
      [Parameter(Mandatory=$true,HelpMessage='Enter the computer
      name to operate on')]
      [string]$computerName
)
```

- Use the Validate* parameter arguments as much as possible for early validation of parameters. E.g. if a file is being passed which must exist use the following:

```
Param
(
      [Parameter(Mandatory=$true,HelpMessage='Enter the file name to
      operate on')]
      [ValidateScript({Test-File -Path $_ -PathType Leaf})]
      [string]$fileName
)
```

  The available validation arguments are:
  - ValidateCount
  - ValidateLength
  - ValidatePattern
  - ValidateRange
  - ValidateScript
  - ValidateSet
  - ValidateNotNull
  - ValidateNotNullOrEmpty
  - ValidateDrive
  - ValidateUserDrive

- When using validation arguments, consider the error output. For some use cases, it can be better to have lax validation in the Param block and write a test in the code that can return a more helpful error message for the user.

- Pass Boolean true or false to scripts as strings so they work correctly. Those strings should then be converted to the native bool type in PowerShell. E.g.:

```
Param
(
    [Parameter(Mandatory=$true,HelpMessage='Whether script
operates in read only mode')]
    [ValidateSet('True','False')]
    [string]$readOnlyParameter
)

[bool]$readOnly = [System.Convert]::ToBoolean(
$readOnlyParameter )
```

The [switch] type cannot be used since that would require support for named parameters.

Functions

- All directives from the “Script Parameters” section should be adhered to except:
    - Optional parameters, since named parameters should be used
    - [Bool] or [switch] native types can be used
- Define your own functions rather than copying and pasting blocks of code
- Use verb-noun format using approved verbs and make the name as meaningful as possible
- When calling functions/cdmlets, always use named parameters without abbreviating the parameter name; do not use positional arguments
- When calling functions/cdmlets, always use the full name of the function/cmdlet, not any alias defined for it, e.g. “Get-ChildItem” instead of “gci” or “dir”
- Use try .. catch blocks around anything that could go wrong. If possible, deal with errors silently if possible without ignoring them. If an error must be reported, do so in as meaningful a way as possible, including any possible remediation action the user could take, and either continue or exit the script depending on the severity/impact of the exception caught
- If the function returns a failed status, follow the approach detailed in the previous bullet
- Every function should have a short explanatory comment block. Unless you feel that it’s appropriate, it’s not necessary to include a full PowerShell format help block, e.g. if the script could be used outside of ControlUp and the function called directly by the user

Variables

- Use meaningful variable names - no abbreviations, e.g. $diskSizeGB not $diskSize, $size or $d
- Use a consistent capitalization strategy; either Pascal case, where the first letter and the first letter of every word in the name is capitalized, or Camel case, which is identical to Pascal case except the very first letter is lowercase, e.g.

```
[string]$nameOfDisk = $null
```

- Add units where applicable to remove any ambiguity, e.g.

```
[int]$diskSizeGB = 0
```

- Always assign default values to variables when declaring if not performing the assignment from a function, cmdlet, etc in the same statement
- When an entity has been closed, freed, disposed, etc, set the variable to $null to help trap further invalid use of that variable, producing weird errors
- Do not use global scope variables unless absolutely necessary - pass as parameters to functions instead
- The abbreviated type name such as “str” for string and “i” for integer can be used to prefix variable names, but be consistent, e.g. [string]$strName or [int]$iDiskSizeGB

Output

- Avoid using unique names in the output; only use unique names where appropriate. Every unique name in the output will result in a new tab, making it harder to compare information. For example:
    - A script that reports the amount of free space on all drives will almost always contain different output for each machine it is run on. If that script is run on five machines, the output will include five tabs. If somebody is copying and pasting the results from the five different tabs having the computername at the top of the output is useful:

```
Machine: XenApp001
Free space C:\ 5.54 Gb
Free space D:\ 12.04 Gb
```

    - A script that reports on the hardware configuration of a machine can be run on several machines to compare the hardware configuration. By not including the name of the machine, all machines with similar hardware will be grouped in one tab in the output.
- When outputting a table with Format-Table, use -AutoSize but set a large output width before any output is produced so that the table does not wrap or become truncated.

```
[int]$outputWidth = 400
# Altering the size of the PS Buffer
$PSWindow = (Get-Host).UI.RawUI
$WideDimensions = $PSWindow.BufferSize
$WideDimensions.Width = $outputWidth
$PSWindow.BufferSize = $WideDimensions
```

- Sort table output on a useful value, such as the largest file first when showing large files, or the last modified date when showing files last modified over a certain period of time.
- Remember that any output written to the error stream will be written to the error window in the output. It’s your choice to write an error to this window, but if you allow the script to continue if it will still provide a (partially) useful result, although you do want to flag errors.