Using PowerShell v5 PSA with Python

Sifos Technologies

Introduction

Sifos PowerSync and PhyView Analyzers are controlled by a software environment named PowerShell PSA. PowerShell PSA constitutes the official API for these instruments and is fully documented in the respective instrument Technical Reference Manual.

PowerShell PSA is implemented as a set of extension functions to the Tcl/Tk scripting language. Tcl/Tk, a popular language in the realm of network testing, is available for Personal Computers running the Microsoft Windows Operating System or the Linux Operating System, as well as for proprietary Unix workstations (such as Sun). The degree of platform and operating system independence renders Tcl/Tk applications highly immune from updates in operating systems and computer hardware.

For those test developers who are working in Python, the **tkinter** package can be used to provide direct access to the complete set of PowerShell PSA Tcl extensions. The **tkinter** package is a "thin" object oriented binary package that is compiled against a specific version of Tcl/Tk.

Note that the **tkinter** package is used with Python version 3.0 and later, and is built against and installed with Tcl/Tk8.6. Earlier versions of Python use a package named **Tkinter** (note the upper case 'T'), which is built against and installed with Tcl/Tk8.5.

Up until version 5.0.01, PowerShell PSA only supported Tcl/Tk8.4 on Windows platforms, and Tcl/Tk8.4 or Tcl/Tk8.5 on Linux platforms. Starting with version 5.1.00, PowerShell PSA supports Tcl/Tk8.4, Tcl/Tk8.5, and Tcl/Tk8.6 (32-bit and 64-bit versions) on both Windows and Linux.

Note that PowerShell PSA is also furnished with a binary API Library, furnished as a **.dll** format (Microsoft Windows) or **.so** format (Linux PC's). This binary API can be used with Python by way of the **ctypes** foreign function library. Examples of how to encapsulate the functions can be provided on request, but accessing PowerShell PSA using **tkinter** is the recommended approach.

The discussions below shows how to use **tkinter** in a text mode Python interpreter. Any of the examples as shown that cause output to ***stdout*** cannot be used in a Python IDLE application, which is a graphical interface, in which the ***stdout*** channel does not exist. While it is technically possible to redirect output from **tkinter** to a Tk widget, that is beyond the scope of this document.

Contents

Using PowerShell PSA with Python 3.x

Python version 3.7.4 was used for the following example statements. The tkinter package installed Tcl 8.6.9.

Loading the PowerShell PSA Tcl Extension in Python 3.x

The PowerShell PSA Tcl extension is loaded by sourcing a runtime configuration file (tclshrc.tcl for Tcl, wishrc.tcl for Tk) from a shell. This is the method of loading the extension defined in the **PSA-3000 Technical Reference Manual version 5**. The following Python statements can be used to instantiate a Tcl interpreter and load the extension using that documented method. This allows the various commands contained in the PowerShell PSA Tcl extension to be executed directly from a Python script:

```
import tkinter
tcl=tkinter.Tcl()
tcl.eval('puts $tcl_version')
rslt=tcl.eval('source "/Program Files (x86)/Sifos/PSA3000/tclshrcAPI_LIB.tcl"')
psaver=tcl.eval('psa_version')
print(psaver)
```

NOTE: an issue was discovered while testing Python integration, when the standard runtime configuration file tclshrc.tcl was used to load the PowerShell PSA extension. When the commands were executed in order as shown above, the command `psaver=tcl.eval('psa_version')` was executed before the connection to the instrument completed, and therefore before the PowerShell PSA extensions had been loaded. This will cause the `print(psaver)` command to fail with an error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'psaver' is not defined.
```

It was determined that the underlying cause was a “user pause” that is performed when using tclshrc.tcl, which gives the user the opportunity to change the IP address of the instrument to connect to. The alternate runtime configuration file tclshrcAPI_LIB.tcl has been modified to set the user pause delay to 0 seconds, which allows the connection to occur immediately, and the Python statements to execute in the order shown, with no additional pacing required.

If you do encounter the error shown above, and determine that some pacing is required on your system, refer to **Techniques to Pace the Connection in Python 3.x** below. Please report this to Sifos (you can email support@sifos,com).

Executing PowerShell PSA Tcl Commands in Python 3.x using tcl.eval

Once the PowerShell PSA Tcl extension has been loaded in the Tcl interpreter which has been instantiated in the Python shell, any of the commands contained in that extension can be executed using **`tcl.eval`**. Here is an example code segment:

```
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```

Once the PowerShell PSA extension has been loaded and the connection to the instrument established, the Python script appears to be correctly paced in terms of the **`tcl.eval`** method not returning until the Tcl command has completed execution (unlike the possible issue with the initial source execution described above).

NOTE: you must use the full PowerShell PSA Tcl command. For example, if you use the short form for the **polarity** command, you will encounter an error:

```
tcl.eval('pol 1,1 neg')
```


The result will be:

```
>>> tcl.eval('pol 1,1 neg')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
_tkinter.TclError: invalid command name "pol"
```

Values that are displayed in the console with the Tcl 'puts' command will not be returned to the Python script by the `tcl.eval` method. Any value that is returned as a Tcl result (i.e. a 'return $somevar' command at the end of a proc) will be returned to the Python script by the `tcl.eval` method. For example, the call:

```
   rslt=tcl.eval('paverage 1,1 stat')
```

will cause the Python variable `rslt` to be set to Tcl result from the average power meter:

```
>>> rslt=tcl.eval('paverage 1,1 stat')
>>> print(rslt)
Slot,Port 1,1
 Average_Power= 10.7 Watts
```

Python variables can be used in Tcl.eval calls. The Python variable needs to be processed with the **tkinter _stringify** method, and used in the statement passed as the argument to **tcl.eval** exactly as shown in the example below. In this example, the address of the slot,port to be queried is contained in the Python variable ***prt***:

```
prt=tkinter._stringify("1,2")
pol=tcl.eval('polarity %(prt)s ?' %locals())
print(pol)
```

```
>>> prt=tkinter._stringify("1,2")
>>> pol=tcl.eval('polarity %(prt)s ?' %locals())
>>> print(pol)
Slot,Port 1,2
Bus_Polarity: NEG
```

Techniques to Pace the Connection in Python 3.x

One way to address a need for pacing is to impose a Python wait after the call to source the runtime configuration file:

```
import time
import tkinter
tcl=tkinter.Tcl()
tcl.eval('puts $tcl_version')
rslt=tcl.eval('source "/ Program Files (x86)/Sifos/PSA3000/tclshrcAPI_LIB.tcl"')
time.sleep(12)
psaver=tcl.eval('psa_version')
print(psaver)
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```

A better way to address a need for pacing is to use a runtime configuration file that provides a way to pace the connection correctly (the source statement returns immediately). This file sets a variable at the very end of the initialization script, and a Tcl proc can be used to source the runtime configuration file, and not return until the script has completed, pacing itself using that variable that is set in the runtime configuration file. A Tcl script named "**psa_init_wait.tcl**" can be provided that handles the initialization in a way compatible with Python (contact Sifos if you need a copy of this script):


```
import tkinter
tcl=tkinter.Tcl()
tcl.eval('puts $tcl_version')
tcl.eval('global env;global myRoot;set myRoot [file normalize
$env(SIFOS_PSA50_ROOT)]')
tcl.eval('source $myRoot/psa_init_wait.tcl')
rslt=tcl.eval('psa_init_wait')
print(rslt)
psaver=tcl.eval('psa_version')
print(psaver)
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```


Using PowerShell PSA with Python 2.x

Python version 2.7.6 was used for the following example statements. The Tkinter package installed Tcl 8.5.2.

Loading the PowerShell PSA Tcl Extension in Python 2.x

The PowerShell PSA Tcl extension is loaded by sourcing a runtime configuration file (tclshrc.tcl for Tcl, wishrc.tcl for Tk) from a shell. This is the method of loading the extension defined in the **PSA-3000 Technical Reference Manual version 5**. The following Python statements can be used to instantiate a Tcl interpreter and load the extension using that documented method:

```
import Tkinter
tcl=Tkinter.Tcl()
tcl.eval('puts $tcl_version')
rslt=tcl.eval('source "/ Program Files (x86)/Sifos/PSA3000/tclshrcAPI_LIB.tcl"')
psaver=tcl.eval('psa_version')
print(psaver)
```

NOTE: an issue was discovered while testing Python integration, when the standard runtime configuration file tclshrc.tcl was used to load the PowerShell PSA extension. When the commands were executed in order as shown above, the command `psaver=tcl.eval('psa_version')` was executed before the connection to the instrument completed, and therefore before the PowerShell PSA extensions had been loaded. This will cause the `print(psaver)` command to fail with an error:

```
---------------------------------------------------------------------------
TclError                                  Traceback (most recent call last)

<ipaython-input-29-78f11350243e> in <module>()
----> 1 print <psaver>
NameError: name 'psaver' is not defined
```

It was determined that the underlying cause was a “user pause” that is performed when using tclshrc.tcl, which gives the user the opportunity to change the IP address of the instrument to connect to. The alternate runtime configuration file tclshrcAPI_LIB.tcl has been modified to set the user pause delay to 0 seconds, which allows the connection to occur immediately, and the Python statements to execute in the order shown, with no additional pacing required.

If you do encounter the error shown above, and determine that some pacing is required on your system, refer to **Techniques to Pace the Connection in Python 2.x** below.

Executing PowerShell PSA Tcl Commands in Python 2.x using tcl.eval

Once the PowerShell PSA Tcl extension has been loaded in the Tcl interpreter which has been instantiated in the Python shell, any of the commands contained in that extension can be executed. Here is an example code segment:

```
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```

Once the PowerShell PSA extension has been loaded and the connection to the instrument established, the Python script appears to be correctly paced in terms of the **`tcl.eval`** method not returning until the Tcl command has completed execution (unlike the possible issue with the initial source execution shown above).

NOTE: you must use the full PowerShell PSA Tcl command. For example, if you use the short form for the **polarity** command, you will encounter an error:


```
tcl.eval('pol 1,1 neg')
```

The result will be:

```
---------------------------------------------------------------------------
TclError                                  Traceback (most recent call last)
<ipaython-input-31-0a08d1e4f346> in <module>()
----> 1 tcl.eval('pol 1,1 neg')

TclError: invalid command name "pol"
```

Values that are displayed in the console with the Tcl 'puts' command will not be returned to the Python script by the `tcl.eval` method. Any value that is returned as a Tcl result (i.e. a 'return $somevar' command at the end of a proc) will be returned to the Python script by the `tcl.eval` method. For example, the call:

```
   rslt=tcl.eval('paverage 1,1 stat')
```

causes the Python variable `rslt` to be set to Tcl result from the average power meter:

```
@TestBed64[c:johns]|10> rslt=tcl.eval('paverage 1,1 stat')
@TestBed64[c:johns]|11> print(rslt)
Slot,Port 1,1
 Average_Power= 10.7 Watts
```

Python variables can be used in Tcl.eval calls. The Python variable needs to be processed with the **tkinter _stringify** method, and the obscure syntax shown in the example below used, where the address of the slot,port to be queried used in the `tcl.eval` call is contained in the Python variable `prt`:

```
prt=Tkinter._stringify("1,2")
pol=tcl.eval('polarity %(prt)s ?' %locals())
print(pol)

@TestBed64[c:johns]|12> prt=Tkinter._stringify("1,2")
@TestBed64[c:johns]|13> pol=tcl.eval('polarity %(prt)s ?' %locals())
@TestBed64[c:johns]|14> print(pol)
Slot,Port 1,2
Bus_Polarity: NEG
```

Techniques to Pace the Connection in Python 2.x

One way to address a need for pacing is to impose a Python wait after the call to source the runtime configuration file

```
import time
import Tkinter
tcl=Tkinter.Tcl()
tcl.eval('puts $tcl_version')
rslt=tcl.eval('source "/ Program Files (x86)/Sifos/PSA3000/tclshrcAPI_LIB.tcl"')
time.sleep(12)
psaver=tcl.eval('psa_version')
print(psaver)
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```

A better way to address a need for pacing is to use a runtime configuration file that provides a way to pace the connection correctly (the source statement returns immediately). This file sets a variable at the very end of the initialization script, and a

Tcl proc can be used to source the runtime configuration file, and not return until the script has completed, pacing itself using that variable that is set in the runtime configuration file. A Tcl script named "**psa_init_wait.tcl**" can be provided that handles the initialization in a way compatible with Python (contact Sifos if you need a copy of this script):

```
import Tkinter
tcl=Tkinter.Tcl()
tcl.eval('puts $tcl_version')
tcl.eval('global env;global myRoot;set myRoot [file normalize
$env(SIFOS_PSA50_ROOT)]')
tcl.eval('source $myRoot/psa_init_wait.tcl')
rslt=tcl.eval('psa_init_wait')
print(rslt)
psaver=tcl.eval('psa_version')
print(psaver)
tcl.eval('alt 1,1 a;polarity 1,1 neg;psa_disconnect 1,1')
rslt=tcl.eval('power_port 1,1 c 3 p 10.7')
print(rslt)
rslt=tcl.eval('paverage 1,1 stat')
print(rslt)
```

