PowerShell PSA - Remote Access Methods

Sifos Technologies

Introduction

Sifos PowerSync and PhyView Analyzers are controlled by a software environment named PowerShell PSA. PowerShell PSA constitutes the only official API for these instruments and is fully documented in *Section 4* of 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 or HP). 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 choose to work in any of the many application environments other than Tck/Tk, Sifos offers facilities, tools, and techniques for gaining remote access to PowerShell API commands. This application note will describe four different scenarios for implementing Remote Access to PowerShell PSA. These are:

1. Accessing PowerShell PSA via TCP Socket Interface
2. Executing PowerShell Scripts Using a Mailbox File Scheme
3. Executing PowerShell Scripts in Batch Mode
4. Calling Functions in the PowerShell PSA API Library

The first three methods all require a high degree of familiarity with PowerShell PSA command syntax for the specific instrument of interest. Experience executing commands and developing scripts in PowerShell PSA will be highly beneficial to those who ultimately use one of these methods in support of an alternative application environment. Working with these methods should make the full range of PowerShell API operations available to application developers.

The fourth method, the API Library, involves usage of special, informally supported library of software that exposes function calls for many, but not all, PowerShell API operations to any language that can integrate a binary library, either in **.dll** format (Microsoft Windows) or **.so** format (Linux PC's). More specifically, one library is furnished for the PowerSync Analyzer family of instruments (PSA, PSL) and a separate library is available for PhyView Analyzers (PVA).

Instruments	Library	Documentation
PSA-3xxx, PSA-12xx, PSL-3xxx, PSA-1200-PL, PSA-2400	PowerShellAPI.dll, libPowerShellAPI.so	PowerShell API Library Ref Manual.pdf
PVA-3xxx	PhyViewAPI.dll, libPhyViewAPI.so	PhyView API Library Ref Manual.pdf

For further information concerning the API Library (method #4), consult the above referenced documentation.

Method 1: Accessing PowerShell PSA via TCP Socket Interface

Socket Server Overview

PowerShell PSA can be controlled remotely by configuring PowerShell PSA to function as a TCP Socket Server. Any client application that can open a socket, write a string to that socket, and perform a blocking read on that socket will be able to remotely access PowerShell PSA. As examples, TCP socket clients interacting with PowerShell have been successfully implemented in C, Python, Tcl, Perl, Visual Basic, and LabView.

When PowerShell PSA is operating as a Socket Server, it will continuously wait to receive PowerShell commands via the TCP socket channel, and when commands arrive, it will execute those commands and respond with either a handshake or a response formed from the exact same data returned to PowerShell. Such data could consist of configuration information, query status, query results, or test results.

In the Socket Server model, each command passed into PowerShell must execute to completion before the next command arrives. This means that client applications must:

1. Read a response to every command issued.
5. Implement their input such that it blocks further execution until a response is provided from the Socket Server (i.e. blocking interface).

Responses to PowerShell commands from the PowerShell Socket Server will be formatted as ASCII strings with user-defined delimiters between data elements and always terminated by an ASCII line feed (0x0A) character. The data element delimiters can be specified when activating the Socket Server to be spaces (" "), semicolons (;), grave accents (`), or carets (^).

PowerShell Socket Server Commands

The following commands in PowerShell PSA may be used to activate and deactivate the TCP Socket Server. These are also described in Section 8 of the Technical Reference Manual for the applicable instrument.

Command	Port	Command Parameters	Returned Parameters
psa_socket_server	<*tcp_port*> <**-space** | **-semicolon** | **-grave** | **-caret** >	This command puts PowerShell PSA into a TCP Socket Server mode where it will automatically receive, process, and respond to PSA commands and queries from a remote client application, either on same host or elsewhere on the network. Query and utility results are passed back through the socket I/O as single lines with user-specified delimiters between elements. Commands will respond with either **COMMAND_OK** or with **PowerShell_ERROR** *Error Message*". Queries will respond with either **RESPONSE** *data* or with PowerShell error message. Special client command **quit** will terminate PowerShell, **psa_server_off** will prevent new server connections, **show port** will return current slot,port value, **show psa** will return currently connected PSA address, and **show error** will return the most recent error message. *tcp_port* TCP port to be assigned to socket server. Default value is **6900**. Range is 1024 to 9999. **-space** Specifies that all response data elements will be separated by a space. Line will terminate with a line feed. This is the default mode. **-semicolon** Specifies that all response data elements will be separated by a semicolon (**;**). Lines will terminate with a semicolon, then a line feed. **-grave** Specifies that all response data elements will be separated by a grave accent (`).Lines will terminate with a grave accent, then a line feed. **-caret** Specifies that all response data elements will be separated by a caret (**^**).Lines will terminate with a caret, then a line feed.	**COMMAND_OK** | **RESPONSE** + delimited ascii string | **PowerShell_ERROR** + delimited ascii string
psa_server_off	Discontinues PowerShell PSA command server such that it will not accept any new connections.		**SERVER_STOPPED**

Socket Client Development Considerations

Any socket client that works with the PowerShell Socket Server must be implemented to allow sufficient time in it's read operations to allow PowerShell PSA to complete the specified command (or task) given. For many commands such as elemental configuration commands, command completion happens in a small fraction of a second so that the subsequent read of the socket channel will almost immediately produce the "COMMAND_OK" handshake. However, many queries, utilities, and fully integrated test commands may require seconds or even minutes of time to complete. It is incumbent on the socket client application to prevent timeouts on read operations until PowerShell resolves and delivers a response.

It is also *essential* that client applications always read for some message in response to each command transmitted. The PowerShell Socket Server will always deliver some form of acknowledgement to every command. This is essential to properly pacing one command at a time to the selected instrument.

If PowerShell encounters any errors in processing an incoming command, it will precede the following response with **PowerShell_ERROR** and follow that with the actual error message developed in PowerShell. For this reason, client applications may want to routinely trap for this particular keyword.

The PowerShell Socket Server can be initiated from a remote application using techniques described later in this application note under **Executing PowerShell Scripts in Batch Mode**. In that case, either the command **psa_socket_server** would be added to the PowerShell Initialization File or a text file using the **.tcl** extension with the command **psa_socket_server** would be stored in the **…\Contrib** directory path.

Users may also desire to edit the PowerShell Initialization File to avoid the timed or menu restricted connection to the default PSA address. This edit is discussed below in the section **Overcoming PowerShell User Connection Interactions**.
The PowerShell Socket Server makes certain special commands available to any socket client application:

- **show port** – Returns current slot and port from PowerShell
- **show psa** – Returns current PSA chassis IP address PowerShell
- **show error** – Returns most recent error message from PowerShell
- **quit** – Terminates the PowerShell session running the Socket Server
- **psa_server_off** – Stops the PowerShell Socket Server from accepting new connections

As with any PowerShell PSA session, the Socket Server shell should not be arbitrarily terminated while commands are executing as this will risk partial command execution at the instrument and the potential for instrument "hang" upon the next attempted instrument connection.

TCP Socket Server Example: Tcl Client

PSA Software includes an example Socket Client application, **psa_tcl_socket_client.tcl**. When this file is sourced into any ordinary Tcl or Wish shell, it enables a simple client connection to an active PowerShell Socket Server running in an existing PowerShell Tcl or PowerShell Wish console window. Note that client connections require the server to be operating before the client connection is made.

Command	Port	Command Parameters	Returned Parameters
psa_socket_client	<*host_name*> < *tcp_port* > < *channel_name* >	This command initiates a client connection to a running PowerShell Socket Server. The connection will block reads until results are provided. Use Tcl **puts** $*channel_name* to send commands and **gets** $*channel_name* to read responses. The connection can be terminated using the standard Tcl command **close** $*channel_name*. *host_name* DNS Host Name or IP Address of server. Default is 'localhost' (same computer). *tcp_port* TCP port to be addressed. This must match the port opened by the PowerShell Socket Server. Default value is **6900**. Range is 1024 to 9999. *channel_name* The desired Tcl variable name for the I/O channel connected to the PowerShell Socket Server. Default name is **psh**.	**COMMAND_OK** | **RESPONSE** + delimited ascii string | **PowerShell_ERROR** + delimited ascii string

In the following example, a Socket Server is started in PowerShell Wish (*see Figure 1*) and a socket client in a Tcl console (*see Figure 2*). From the Tcl console, a chassis connection is performed, then an auto-discover of PSE characteristics on Slot 1, Port 1 (i.e. ALT, Polarity, MPS method). Following that, an emulated PD Class 3 power-up to 14.5 watts is performed followed by a power measurement, then a disconnect shutdown. Finally a PSE Conformance Test, **det_v**, is executed on that same slot and port.

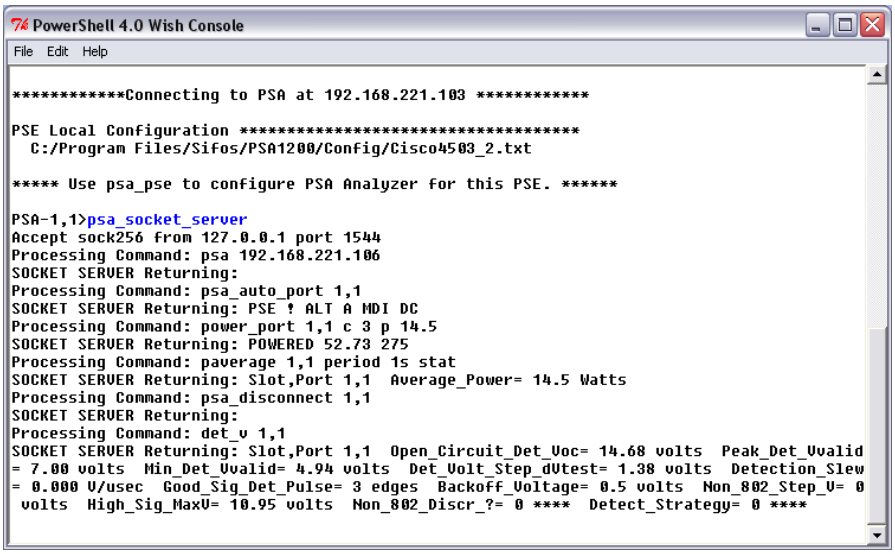

Figure 1: PowerShell Running Socket Server with Default Settings

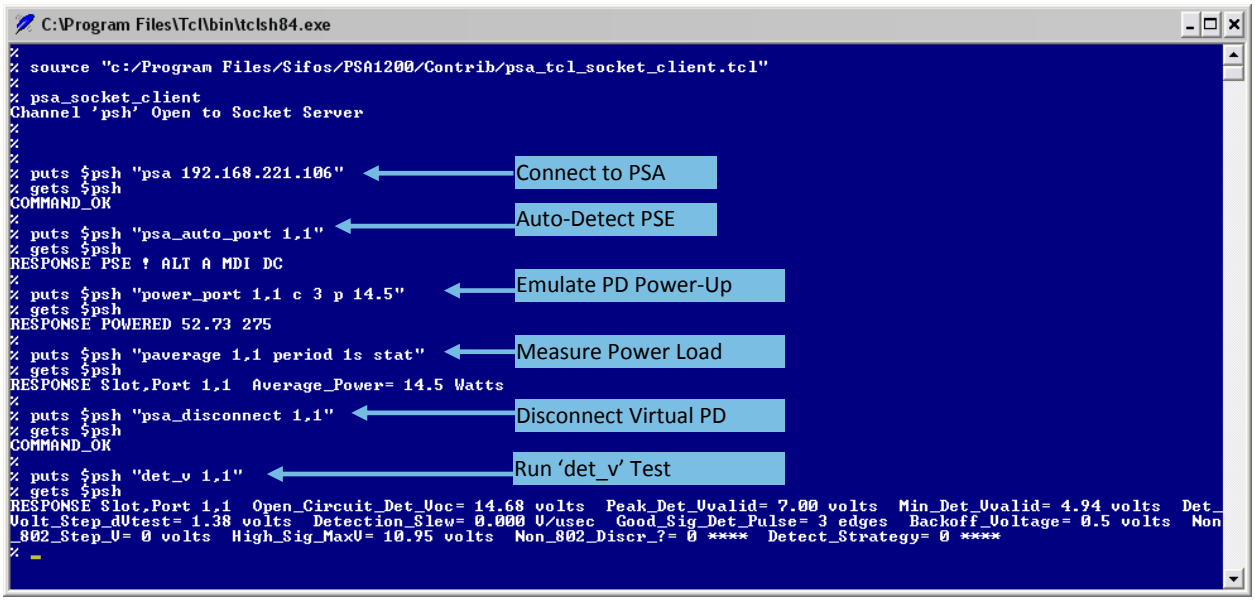

Figure 2: Tcl Shell with Socket Client Running with Default Settings

Sifos Application Notes with specific examples of the TCP Socket Interface are available for the following application environments:

- Microsoft Visual Basic 6 (PowerShell Socket Client - Visual Basic 6 Application.pdf)
- National Instruments LabView (PowerShell Socket Client - LabView Application.pdf)

Method 2: Executing PowerShell Scripts Using a Mailbox File Scheme

Mailbox File Scheme Overview

PowerShell PSA can be placed into a mode to facilitate inter-process communication using pre-designated text files. In this scheme, text files are utilized for transmitting one or more commands to PowerShell and then acknowledging command completion by PowerShell.

This scheme is generally most effective when test applications or utilities are developed inside of PowerShell PSA, including any required result processing and reporting facilities, then those test applications are invoked or managed remotely.

Virtually any command (or script name) that can be typed into PowerShell can be passed in via the command file. All commands are constructed in accordance with PowerShell PSA syntax rules described in Section 4 of the Technical Reference Manual. Sequences of commands may be transmitted in a single file provided they are all delimited by braces ({command}), line feeds (command**LF**command) , or semicolons (command;command;command). Command completion is then acknowledged through a response file with either **DONE** or **ERROR** status.

Activating and Passing Commands to PowerShell

Prior utilizing the mailbox file scheme, either a **PowerShell Tcl** or **PowerShell Wish** session is opened on the host computer. This can be done manually from the desktop or remotely from another application.

If opened manually, the command echo mode is activated with the command **psa_echo**. There are no arguments to this command.

To open PowerShell Tcl (or Wish) remotely and activate the echo mode, the PowerShell Initialization File (**tclshrc.tcl** for PowerShell Tcl and **wishrc.tcl** for PowerShell Wish) must be edited using a text editor or the AnyEdit editor included with PSA software. This edit simply requires the "un-commenting" of the **psa_echo** command at the end of this file, that is, removal of the pound (**#**) character in front of the **psa_echo** command. Users may optionally desire to edit the PowerShell Initialization File to avoid the timed or menu restricted connection to the default PSA address. This edit is discussed below in the section **Overcoming PowerShell User Connection Interactions**.

The choice of PowerShell Tcl or PowerShell Wish is mostly arbitrary, although PowerShell Wish offers the capability to produce graphical PSA-3000 waveform traces and PVA-3000 PSD traces should those be of interest.

Once the PowerShell command "server" is operational, the remote application must work with two text files:

File	Purpose
psa_command.txt	Pass Commands to PowerShell
psa_response.txt	Receive Command Status from PowerShell

These files are located in the PSA environment directory that is located as follows:

Operating System	Location
Microsoft Windows 7, Vista	\Users\Public\Sifos\PSA1200\Config\env\
Microsoft Windows XP, 2000	\Program Files\Sifos\PSA1200\Config\env\
Linux	$HOME/Sifos/PSA1200/Config/env/

The client application that is passing commands into PowerShell is responsible for creating instances of the **psa_command.txt** file that include PowerShell commands. When this file is created and contains actual data, that is **psa_command.txt** file size is greater than zero bytes, the PowerShell "server" in echo mode will automatically and sequentially process commands from that file. It will then delete the **psa_command.txt** file and upon completion of the command execution, and will create an instance of the **psa_response.txt** file for the client application to read. That file will simply provide the value "**DONE**" or "**ERROR**". The client application would then delete the **psa_response.txt** file so that it can resume monitoring for this file following the next command transmission. **psa_response.txt** will only appear AFTER the status value has been recorded.

If the client application creates **psa_command.txt** and is able to load data to it before closing (or releasing) that file, there is the possibility of a race condition where the PowerShell server reads it before it is completed. This problem can be avoided in one of two ways:

1. Create **psa_command.txt** in a manner where actual data is not recorded until the file is closed.
6. Create **psa_command.txt** with an alternative name, then simply rename the file to **psa_command.txt** after the file has been saved.

Formatting Commands in psa_command.txt

One or more commands may be stored in any given instance of the **psa_command.txt** file. The following table describes the valid options for formatting commands within this file.

Method	Single Command	Command Sequence
No Delimiters	**power_port 1,1 c 3 p 10** (as entered to PowerShell)	
Brace Delimiters	**{alt 1,1 a}**	**{alt 1,1 a}{passive 1,1 r 25 c 0}** *or* **{alt 1,1 a} {passive 1,1 r 25 c 0}** *or* **{alt 1,1 a}** **{passive 1,1 r 25 c 0}**
Semicolon Delimiters		**alt 1,1 a;passive 1,1 r 25 c 0** *or* **alt 1,1 a; passive 1,1 r 25 c 0** *or* **alt 1,1 a;** **passive 1,1 r 25 c 0**
Line Feed Delimiters		**alt 1,1 a** **passive 1,1 r 25 c 0** **port 1,1 connect**

Recovering Results

Any measurement or test results (configurations, statuses, measurements, test data) produced as a result of commands passed into PowerShell will remain within PowerShell unless those PowerShell commands are already designed to export results, for example, outputting PSE Conformance Test Suite results or PHY Performance Test Suite results to a user specified text file. For this reason, the mailbox file scheme is mostly intended to execute PowerShell PSA scripts that have been written to output measurements and test data to files or other devices such that the client application can recover that test data.

The PowerShell Socket Server described earlier would make better sense if the desire is to have external applications that interact with instruments to configure resources and make measurements.

Mailbox File Scheme Example: Tcl Client

In the following example, a PSA-3000 test port will be configured to a particular PSE from a remote Tcl shell, and PSE Conformance Tests will be sequenced to a user specified data file. The client is an ordinary Tcl shell in this example.

First, PowerShell Wish is opened and put into command echo mode (*see Figure 3*).

A Tcl shell is then opened (*see Figure 4*) and commands are sequenced to first configure the PSA test port(s) to the PSE (**psa_auto_port**), and then to sequence four PSE conformance tests (**sequence**) with the report going into a command-specified file (c:\myFile.txt).

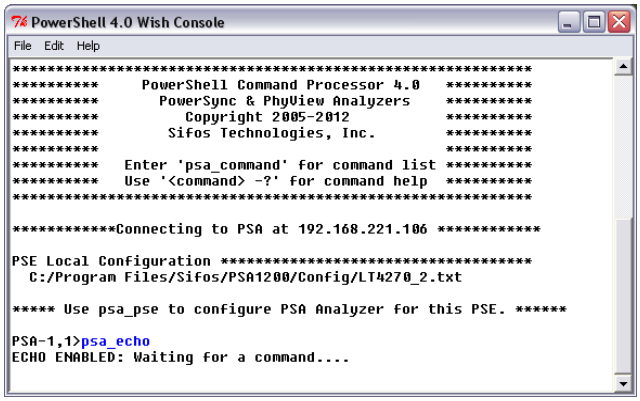

Figure 3: PowerShell Wish Operating in Echo Mode

The sequential steps required to pass commands and read acknowledgements is depicted in Figure 4. In summary, for each command or group of commands, these steps include:

1. Open **psa_command.txt** file and write the command information.
2. Monitor for **psa_response.txt** file to appear – specify an appropriate time limit based upon command.
3. Read and delete **psa_response.txt** file.
4. Process any data files created by PowerShell scripts that are executed.

In this particular example, the sequence command was instructed to develop a text result file stored at **c:\myFile.txt** (*see Figure 5*).

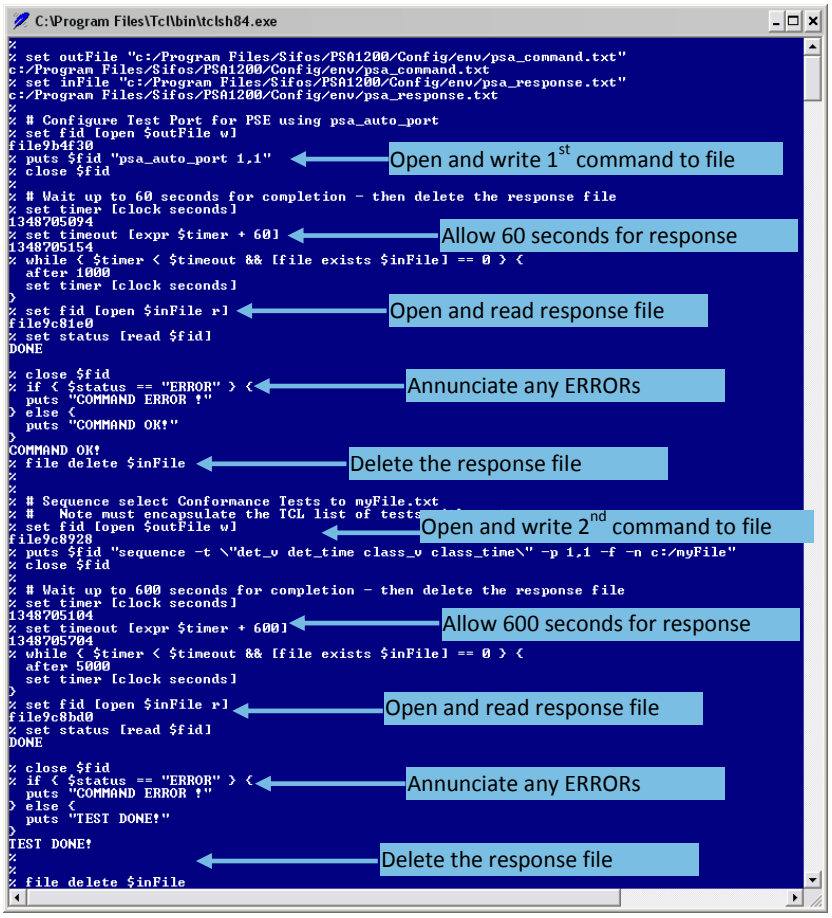

Figure 4: Tcl Shell Passing Commands and Reading Status

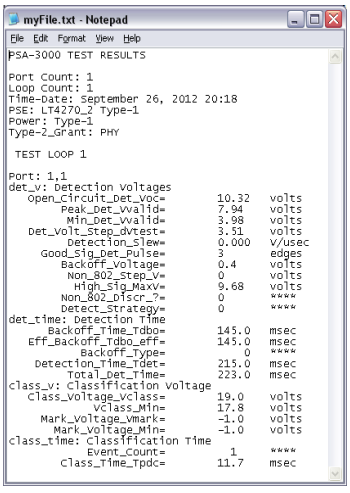

Figure 5: Test Report from **sequence**

Method 3: Executing PowerShell Scripts in Batch Mode

When PowerShell PSA is opened under control of an external application, there are three methods by which pre-written PowerShell scripts can be automatically executed:

1. Batch Mode Method 1: PSA1200 **Contrib** Directory
7. Batch Mode Method 2: Edit PowerShell Initialization File to Add Script Call
8. Batch Mode Method 3: PowerShell Launch Program Command Arguments

The first batch mode method simply requires that the user script file be stored in **Contrib** directory associated with PSA software. All **.tcl** files stored within this directory (excluding any sub-directories) are automatically sourced into PowerShell when PowerShell opens. User scripts that are not structured as Tcl commands (i.e. **proc {}**) will then execute upon loading into PowerShell. The Contrib directory is located as follows:

Operating System	Location
Microsoft Windows 7, Vista	\Users\Public\Sifos\PSA1200\Contrib\
Microsoft Windows XP, 2000	\Program Files\Sifos\PSA1200\Contrib\
Linux	$HOME/Sifos/PSA1200/Contrib/

The second batch mode method involves editing the PowerShell Initialization File (**tclshrc.tcl** for PowerShell Tcl and **wishrc.tcl** for PowerShell Wish) as described above under **Activating and Passing Commands to PowerShell**. In this method, the user script is implemented as a Tcl "**proc {}**" and saved in the **\Contrib** directory. Then, the PowerShell Initialization File is edited to add the call to the desired user script (*or proc name*) at the end of that file, much like activating the **psa_echo** call for mailbox file command passing described earlier.

Both of these batch mode methods are only viable for scenarios where there is *only a single user script* that would ever execute since there is no means for selecting which scripts load and execute. They also have the drawback that when PowerShell is opened manually, the script will automatically execute.

The third batch mode method involves passing a script name and optionally and output file name when executing the PowerShell launcher program. This method allows both the selection of a particular script and the ability to route PowerShell output to a user-specified file. With this method, the script files should either not use the **.tcl** extension, or should not be placed in the PSA software **\Contrib** directory so that automatic script execution is avoided.

Batch Mode Execution in Microsoft Windows

PowerShell Tcl and PowerShell Wish are each opened using launcher programs that can accept optional arguments defining a PowerShell script file to source and an output file for channeling data. This means that a script developed for PowerShell can be invoked from an outside application and the data developed by that script can be recovered by that outside application. The following table shows the launcher programs and argument structures.

PowerShell Type	Executable	1st Argument	Redirect	2nd Argument
PowerShell Tcl	PowerShell TCL.exe	Script File (e.g. myScript.tcl)	>	Data File (e.g. myResults.txt)
PowerShell Wish	PowerShell Wish.exe			

Depending upon the version of Microsoft Windows, the PowerShell PSA launchers are found at the following directory paths:

Operating System	Location	Initialization File
Microsoft Windows 7 (64 bit)	\Program Files (x86)\Sifos\PSA1200\	**tclshrc.tcl** or **wishrc.tcl**
Microsoft Windows 7 (32 bit), Vista, XP, 2K	\Program Files\Sifos\PSA1200\	

The following Windows Command Shell sequence will launch PowerShell Tcl and automatically invoke a script stored in c:\Program Files\Sifos\PSA1200\Contrib\No_Source**myScript.tcl**, redirecting output to c:**myResults.txt**.

```
REM Move to launcher directory
cd c:\Program Files\Sifos\PSA1200

REM Launch PowerShell Tcl to run myScript.tcl and output to myResults.txt
"PowerShell Tcl.exe" Contrib\No_Source\myScript.tcl > c:\myResults.txt
```

Overcoming PowerShell User Connection Interactions

When PowerShell Tcl or PowerShell Wish launches, it automatically sources (or executes) a PowerShell Initialization File also found in the same directory path as the launcher program. The PowerShell Initialization File is what converts the otherwise ordinary Tcl or Wish shell into PowerShell Tcl or PowerShell Wish.

One of the functions of the PowerShell Initialization File is to manage the initial instrument connection established when the shell opens. PowerShell Tcl normally presents a user prompt to override a connection to the most recently connected chassis, then after a period of time, if no response is entered, it continues on and completes the connection to that instrument. PowerShell Wish presents a graphical connection dialog menu and will wait forever for the user to finalize the instrument connection via that dialog.

When PowerShell is invoked remotely, there will be a need to bypass the user prompt in PowerShell Tcl or the connection dialog window in PowerShell Wish. This is easily accomplished with a minor edit to the appropriate PowerShell Initialization File as described in the following table.

<table>
<tr><th>PowerShell Type</th><th>Initialization File</th><th>Location</th><th>Edit</th></tr>
<tr><td>PowerShell Tcl</td><td>tclshrc.tcl</td><td rowspan="2">Same as Launcher Program</td><td><pre>
Initial Connection Control
Set psaConnectPause to 0 to bypass initial connection address prompt and automatically connect
to most recently connected PSA
Set psaConnectPause to between 2 and 60 seconds set the prompt waiting time before
connection to most recenlty connected PSA automatically occurs.
NOTE!!!!
The connection dialog should NOT be bypassed where multiple PSA's are utilized by multiple
users on the same network unless PSA's are fully dedicated to individual users!
set psaConnectPause 0
if {$psaConnectPause > 0 && $psaConnectPause < 2} {
  set psaConnectPause 2
}
if {$psaConnectPause > 60} {
  set psaConnectPause 60
}
</pre></td></tr>
<tr><td>PowerShell Wish</td><td>wishrc.tcl</td><td><pre>
PowerShell Wish Control of Initial Connection Dialog
Set psaConnectPause = 0 to bypass initial connection dialog and force PowerShell Wish
to open to most recent connection
Set psaConnectPause = 1 to force PowerShell Wish to wait indefinitely for user response
to PSA Connection Dialog
NOTE!!!!
The connection dialog should NOT be bypassed where multiple PSA's are utilized by multiple
users on the same network unless PSA's are fully dedicated to individual users!
set psaConnectPause 0
</pre></td></tr>
</table>

Batch Mode Example – Windows Command Shell

In the following example, a PowerShell PSA test script (**myScript.tcl**) will be launched from an ordinary Windows command shell, **cmd.exe** (*see Figure 6*). This script will measure port voltages on a PSE connected to ports 1,1 – 2,2 of the PowerSync Analyzer located at IP address 192.168.221.106. Results will be routed to the file **c:\myResults.txt**. Prior to launching this command, the **tclshrc.tcl** Initialization File was modified to bypass the user connection dialog.

```
# myScript
psa 192.168.221.106
alt 99,99 A
polarity 99,99 neg
foreach port "1,1 1,2 2,1 2,2" {
  set status [power_port $port c 0]
  append Vport "Port $port: [lindex $status 1] volts\n"
  psa_disconnect $port
}
puts $Vport
```

myScript.tcl is stored in the directory path **…\Contrib\No_Source** so that it will not automatically execute whenever PowerShell opens.

Upon completion, the results file **c:\myResults.txt** contained PowerShell Tcl output (*see Figure 7*).

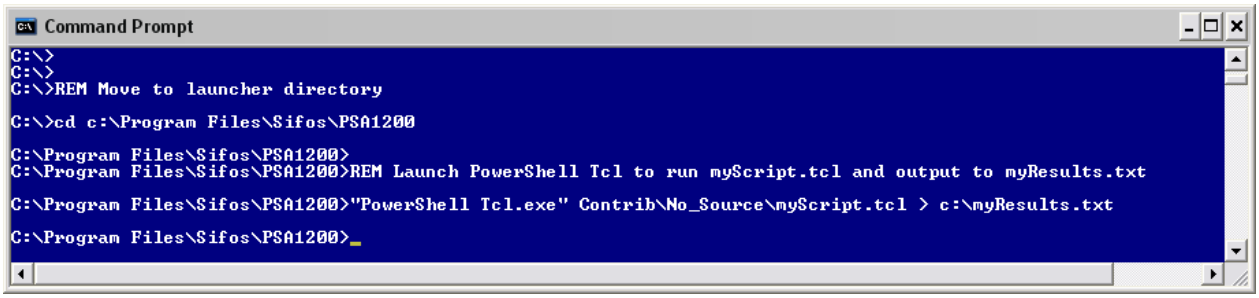

Figure 6: Windows Command Shell Launching PowerShell Tcl with myScript.tcl

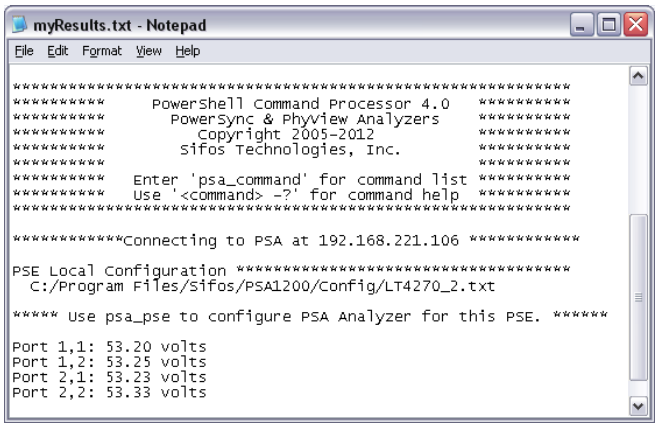

Figure 7: myResults.txt With PowerShell Tcl Output

Batch Mode Execution in Linux

The Linux installation of PSA Software includes shell scripts that may be utilized to launch PowerShell Tcl and PowerShell Wish.

PowerShell Type	Launcher Script	Default Location*
PowerShell Tcl	PowerShell_TCL.sh	**$Home/bin** and **/usr/local/Sifos/PSA1200/**
PowerShell Wish	PowerShell_WISH.sh	

* Linux/Unix Installations may be routed to other locations depending upon user preferences and administrative permissions.

As shell scripts, these may be invoked from Linux command shells (e.g. **bash** shell). However, unlike the Windows launchers above, they will not accept command arguments to specify particular scripts and output routing. This leaves only batch mode methods 1 and 2 described earlier for Batch Mode Execution of PowerShell scripts.

Since Tcl or Wish shells underlie PowerShell PSA, the PowerShell PSA shell requires Tcl/Tk to be installed on the host computer such that Tcl and Wish shells will load with their binary executable commands (e.g. **tclsh8.4** or **wish8.4**). The following code example will then initiate PowerShell Tcl from the native Linux command shell:

```
cd $HOME/bin
PowerShell_TCL.sh
```

To automatically execute a user script, that script must be located in the **…/Contrib/** directory path. If the script is not structured as a Tcl command (i.e. **proc {}**), it will run when PowerShell starts (*see batch mode method #1*). Otherwise, the appropriate PowerShell Initialization File must be edited to incorporate the script call at the end of that file (*see batch mode method #2*).

On a Linux system, the PowerShell Initialization Files are located at **$HOME/Sifos/PSA1200**. When the launch script (e.g. **PowerShell_TCL.sh**) is executed, the appropriate Initialization File is automatically (and temporarily) copied into the **$HOME** directory so that Tcl (or Wish) will incorporate it to run PowerShell.

File Permissions & Updates

Users should be aware that when PSA software is installed, the PowerShell Initialization Files **tclshrc.tcl** and **wishrc.tcl** have read-only permission. To modify them, their permissions must be changed.

Also, when software upgrades are installed, users should assume that any modifications made locally to the PowerShell Initialization Files will be lost, so it is important that copies of those modified files are maintained in other directories.

