Application Note Remote Control of PowerShell Tcl

Introduction

The Sifos PowerSync Analyzer is controlled by a software environment named PowerShell PSA. PowerShell PSA is implemented as a set of extension functions to the Tcl scripting language, which was chosen to provide cross platform portability. Tcl is available on Personal Computers running the Microsoft Windows Operating System or the Linux Operating System, as well as on proprietary Unix workstations (such as Sun or HP).

The Application Programming Interface (API) for the Sifos PowerSync Analyzer is the set of PowerShell PSA Tcl extension functions (referred to below as PowerShell Tcl). The PowerSync Analyzer instrument does not implement a standalone Command Line Interface (CLI) – all control is performed via a binary protocol communicated to the instrument via LAN, using TCP port 23.

NOTE: any attempt to access the instrument directly, testing for a CLI, for example using telnet, will most likely leave the instrument in an odd state, requiring the instrument to be re-booted.

The different ways that PowerShell can be controlled remotely are:

1) Executing PowerShell commands using a 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

You should notice a common thread to remote control methods 1 through 3 – they ALL require that you understand the PowerShell Tcl scripting language, and how the PowerSync Analyzer behaves in response to various commands.

Method 4 uses functions published in a binary library (a .dll on Windows platforms, or a .so on Linux platforms). Use of the API library is covered in the ***PowerShell API Library Reference Manual***.

This application note is written with the assumption that the user has an understanding of the Tcl scripting environment, and the PowerShell Tcl functions that comprise the API. If more information is needed regarding the Tcl scripting environment, a good reference is ***Practical Programming in Tcl and Tk***, by Brent B. Welch, as well as a wealth of examples that can be found on the World Wide Web. For PowerShell Tcl, refer to the ***PowerSync Analyzer PSA-3000 Technical Reference Manual***. The API is documented in Chapter 4 *PowerShell Scripting Environment*.

In all cases, it is **strongly** recommended that the user prototype any commands that they intend to use in a PowerShell Tcl Console window (or PowerShell Wish Console window), build up a successfully functioning script, and then utilize their automation environment to execute those statements to remotely control a PowerSync Analyzer.

Method 1 Remote Control of PowerShell Tcl Using a Socket Interface

The General Technique

PowerShell PSA can be controlled remotely by configuring a shell to function as a 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 control PowerShell PSA. At this point, clients have been successfully implemented in C, Python, Tcl, Perl, Visual Basic, and LabView.

The technique involves running a concise script in a PowerShell Tcl session that causes it to function as a socket server. The Tcl session opens a socket on a defined TPC/IP port, configures it to function as a server, and then performs a blocking receive, waiting for data to arrive.

The client opens a socket to the defined TCP/IP port, and writes PowerShell PSA Tcl command strings to the socket, which will be read by the server. When a command string is received by the PowerShell server, it executes that command string using the Tcl '**eval**' command. When the command has been processed, the server will write the standard Tcl result of that command to the socket, using the Tcl '**puts**' command. The string that will be read from the socket will be terminated by a newline character (ascii 0x0A). Note that some commands do not produce any output, which causes the Tcl result to be an empty string. The result from **every** command will be written to the socket by the server, and **must** be read by the client side, in order to flush the socket, and keep the client synchronized with the server. As stated above, this **includes** any empty string written on the socket by the server, if that is the result of the last command executed (this is discussed further below, in the *Client/Server Example* section of this document). An empty string as read by the client will contain a single newline character.

If the command written to the server causes an error, that error will be trapped, and the error message returned in a string formatted: "COMMAND ERROR: $errMsg", where $errMsg will be replaced by the actual text of the error message returned by the Tcl shell.

NOTE: the protocol that **must** be employed by the client when using PowerShell Tcl as a socket server is to **always** perform a write followed by a read. These operations must always be performed as a pair. The client should not write any command to the socket until the server has responded to the last command written. The **only** exception to this pattern is the "quit" command, which it used to terminate the server. The server cannot write a response on the socket after receiving the "quit" command, since it is closing the socket. The string "quit" is not a valid PowerShell Tcl command – the server code tests for "quit" in every string received, before submitting the string to Tcl for evaluation.

Timing Considerations

Many PowerShell Tcl commands that configure instrument settings will execute rapidly (35 – 70 msec), such as '**alt**' or '**polarity**'. However, commands that perform measurements that are dependent on the behavior of the Device Under Test (DUT) will vary in the length of time required for completion. For example, the command '**power_port**' configures the active load and port connect setting, and then performs measurements to determine the state of the PSE port. If the PSE does not successfully power up the port, the command can take in excess of 35 seconds to complete. There are certain PSEs that have been discovered to exhibit very long back off times (~10 seconds), which has the potential to cause certain Conformance Test commands to take minutes to complete. A Conformance Test sequence can take 10s of minutes to complete, depending on the number of ports being tested.

A rigorous implementation of a client application should include the ability to timeout if the server does not write a response on the socket in a defined time period. The length of this time period is directly a function of the behaviors of the PSE being tested, and the commands being sent to the server. The principal goal of enforcing a timeout is to prevent an automation client from hanging infinitely if a problem occurs, either with the socket connection or the server.

The user will need to determine the correct value to use for a timeout period. Tcl provides a native command that can be used to time command execution. Here is an example of using the Tcl '**time**' command to measure how long a Socket write/read takes for a '**power_port**' command that terminates after the PSE port does not successfully power on:

```
%set et [time {puts $s "power_port 1,1 c 0" ; set resp [gets $s] }]
35213746 microseconds per iteration
```

For consistent behavior, the client should normally **not** terminate in the middle of a transaction, or kill the server window while in the middle of a transaction. If the server window is abnormally terminated in the middle of a transaction, there is a chance that a partial message will still be present in the PSA-1200/PSA-3000 Controller, which will then cause the next attempted connection from a PowerShell Tcl session to the PowerSync Analyzer to fail. The cause of the connection failure is due to a badly formed packet being received by the PC (which is "corrupt" as far as PowerShell Tcl is concerned). If this occurs, a subsequent connection attempt should function normally. (This same problem can occur if using a PowerShell window in standalone mode, and terminating that window abnormally while a command is in the middle of executing. It is not specific to using a socket to control PowerShell Tcl).

If a timeout period is properly enforced, and the client application abnormally terminates, and closes the socket due to the timeout occurring, then the connection problem described above may occur. It is recommended that the client application trap on an error with the first attempted connection, and re-try the connect operation if the first connection indicates an error.

Client/Server Example

The following example code provides the means for the user to experiment with the PowerShell Tcl Socket Interface method. Both the client and server sides of the interface described below can be implemented with the software installed for the PowerSync Analyzer.

The code segment listed in Figure 1 defines the Tcl script that you would load and execute in a PowerShell window to cause it to function as a socket server. This script must be running in a PowerShell Tcl window before a client attempts to connect. See notes below the listings regarding how to load PowerShell Tcl and start the socket server.

The server Tcl script is provided in the file:

~Sifos/PSA1200/Contrib/ socket_server.txt

This server returns the exact value that each Tcl command returns.

An alternative server example is also provided (starting with PowerShell versions after 3.5.07), in the file:

~Sifos/PSA1200/Contrib/socket_server_delimited.txt

This script inserts a ";" delimiter between each list element returned by a Tcl command, and when the Tcl command's result is an empty string, returns "EMPTY_STRING". All responses (for commands that do not cause a Tcl execution error) are predicated with the string "COMMAND SUCCESS - result:". This server script is shown in Figure 2.

The code segment listed in Figure 3 contains example Tcl code that shows how you would create a socket from a generic Tcl session, allowing it to function as the client application.

The client Tcl script is provided in the file:

~Sifos/PSA1200/Contrib/ socket_client.txt

The server shell will attempt to process each command received until the client writes the command "quit" to the socket. The server will then terminate, closing the socket, and will not write any response to the client.

```
proc Echo_Server {port} {
  global echo
  set echo(main) [socket -server CmdAccept $port]
}
proc CmdAccept {sock addr port} {
  global echo
  puts "Accept $sock from $addr port $port"
  set echo(addr,$sock) [list $addr $port]
  fconfigure $sock -buffering line
  fileevent $sock readable [list ProcessCmd $sock]
}
proc ProcessCmd {sock} {
  global echo
  global resp
  if {[eof $sock] || [catch {gets $sock line}]} {
    #end of file, or abnormal connection drop
    close $sock
    puts "Close $echo(addr,$sock)"
    unset echo(addr,$sock)
  } else {
    if {[string compare $line "quit"] == 0} {
      #prevent new connections
      #existing connections stay open
      close $echo(main)
      puts "Close $echo(addr,$sock)"
      unset echo(addr,$sock)
      exit
    }
    puts "processing cmd: $line"
    set err [catch {set resp [eval $line]} errMsg]
    if { $err == 1 } {
      puts $sock "COMMAND ERROR: $errMsg"
      puts "COMMAND ERROR: $errMsg"
    } else {
      puts "returning: $resp"
      set jresp [join $resp]
      puts $sock $jresp
    }
  }
}
Echo_Server 6900
vwait forever
```

Figure 1

```
proc Echo_Server {port} {
  global echo
  set echo(main) [socket -server CmdAccept $port]
}
proc CmdAccept {sock addr port} {
  global echo
  puts "Accept $sock from $addr port $port"
  set echo(addr,$sock) [list $addr $port]
  fconfigure $sock -buffering line
  fileevent $sock readable [list ProcessCmd $sock]
}
proc AddDelimeters {rawList} {
  set resp ""
  for {set i 0} {$i < [llength $rawList]} {incr i} {
    append resp [lindex $rawList $i]
    append resp ";"
  }
  set resp [string map {"\n" ""} $resp]
  return -code ok $resp
}
proc ProcessCmd {sock} {
  global echo
  global resp
  if {[eof $sock] || [catch {gets $sock line}]} {
    #end of file, or abnormal connection drop
    close $sock
    puts "Close $echo(addr,$sock)"
    unset echo(addr,$sock)
  } else {
    if {[string compare $line "quit"] == 0} {
      #prevent new connections
      #existing connections stay open
      close $echo(main)
      puts "Close $echo(addr,$sock)"
      unset echo(addr,$sock)
      exit
    }
    puts "processing cmd: $line"
    set err [catch {set resp [eval $line]} errMsg]
    if { $err == 1 } {
     puts $sock "COMMAND ERROR: $errMsg"
     puts "COMMAND ERROR: $errMsg"
    } else {
     puts "returning: $resp"
     set jresp [AddDelimeters $resp]
     if {[string len $jresp] == 0} {
       puts $sock "COMMAND SUCCESS - result: EMPTY_STRING"
     } else {
       puts $sock "COMMAND SUCCESS - result: $jresp"
     }
    }
  }
}
Echo_Server 6900
vwait forever
```

Figure 2

```
proc Echo_Client {host port} {
   set s [socket $host $port]
   fconfigure $s -buffering line
   return $s
}
set s [Echo_Client localhost 6900]
```

Figure 3

The PowerShell Tcl socket server is executed in a PowerShell Tcl Console window. A technique that can be used to automate starting a PowerShell Tcl server is to modify the initialization file **c:\Program Files\Sifos\PSA1200\tclshrc.tcl**, adding a statement at the end of the file to source a script that contains the code listed in Figure 1. For example, with the code from Figure 1 stored in the file c:\Program Files\Sifos\PSA1200\Contrib\socket_server.txt, the following statement would be added as the **last** statement in the modified **tclshrc.tcl**:

```
source  "$psaPath/Contrib/socket_server.txt"
```

Note that this statement must come after the code that initializes PowerShell Tcl:

```
# Initialize PowerShell
if { [file exists $psaPath/psaInit.tbc] } {
        source $psaPath/psaInit.tbc
} else {
        source $psaPath/psaInit.tcl
}

source  "$psaPath/Contrib/socket_server.txt"
```

A PowerShell Tcl Console can be programmatically started by a client by executing the statement:

```
c:\Program Files\Sifos\PSA1200\PowerShell Tcl.exe
```

This "launcher" program will establish a Tcl interpreter, and will load ~ PSA1200\tclshrc.tcl.

NOTE: there is a space between "PowerShell" and "Tcl". Depending on the execution context, you may need to enclose the path in quotes, to deal with the embedded white space.

NOTE: it is not possible to execute PowerShell Tcl.exe (or tclsh.exe) with the startup script provided as a command line argument. When executed using that form of command, the shell will start, source the file specified on the command line, evaluate each statement, and will exit when all lines in the file have been processed. The Tcl shell must be started in interactively, which can be done only by executing the .exe with no command line arguments.

Some example PowerShell commands are shown below in Listing 1, followed by a snapshot of the commands as executed in the client window, in Listing 2, and finally a snapshot of the messages that would be displayed in the server window, in Listing 3. The response that will be sent by socket_server_delimited.txt to a command which returns an empty Tcl result is shown in Listing 4.

The example was investigated using a PSA-3000 with (3) PSA-3102 blades installed, in slots 1, 2, and 3. The host was a PC running Windows XP. The Client window was a generic Tcl shell, started with the menu command:

Start -> All Programs -> Tcl -> tclsh

On a machine that has ActiveState Tcl loaded, this would be a path similar to:

Start -> All Programs -> ActiveState ActiveTcl 8.4.12 -> Tclsh84

Listing 1

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Example PowerShell commands
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# query the chassis inventory
puts $s "psa_config"
set resp [gets $s]

# connect the port, power up PSE
puts $s "power_port 2,1 c 0"
set resp [gets $s]

# read the status indicators
puts $s "pstatus 2,1"
set resp [gets $s]

# measure the current
puts $s "idcaverage stat"
set resp [gets $s]

#measure the voltage
puts $s "vdcaverage stat"
set resp [gets $s]

# increase the load
puts $s "iload 2,1 i 200"
set resp [gets $s]
```

```
# measure the current
puts $s "idcaverage stat"
set resp [gets $s]

# measure the current, return only the value (none of the surrounding text)
puts $s "lindex \[idcaverage stat\] 3"
set resp [gets $s]

# increase the current to a point that will cause the PSE to shutdown
puts $s "iload 2,1 i 450"
set resp [gets $s]

# measure the port voltage
puts $s "vdcaverage stat"
set resp [gets $s]

# measure the current
puts $s "idcaverage stat"
set resp [gets $s]

# perform an orderly shutdown of the port
puts $s "psa_disconnect"
set resp [gets $s]

# this is a command that will cause an error
puts $s "alt 4,2 ?"
set resp [gets $s]

# close the server session
puts $s "quit"
```

Listing 2

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Example client session, running in a generic Tcl window.
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C:\Documents and Settings\johns>tclsh84
% proc Echo_Client {host port} {
  set s [socket $host $port]
  fconfigure $s -buffering line
  return $s
}
% set s [Echo_Client localhost 6900]
sock1748
% # query the chassis inventory
% puts $s "psa_config"
% set resp [gets $s]
0,0 PSA3000_Controller_01 3.00 1,1 PSA3102_Test_Blade_01 3.01 1,2 PSA3102_Test_Blade_01 3.01 2,1
PSA3102_Test_Blade_01 3.01 2,2 PSA3102_Test_Blade_01 3.01 3,1 PSA3102_Test_Blade_01 3.01 3,2
PSA3102_Test_Blade_01 3.01 4,1 Empty . 4,2 Empty . 5,1 Empty . 5,2 Empty . 6,1 Empty . 6,2
Empty . 7,1 Empty . 7,2 Empty . 8,1 Empty . 8,2 Empty . 9,1 Empty . 9,2 Empty . 10,1 Empty
. 10,2 Empty . 11,1 Empty . 11,2 Empty . 12,1 Empty . 12,2 Empty .
%
% # connect the port, power up PSE
% puts $s "power_port 2,1 c 0"
% set resp [gets $s]
POWERED 48.90 140
%
% # read the status indicators
% puts $s "pstatus 2,1"
% set resp [gets $s]
Detection: ON Powered: ON Arming: OFF Aux_LED: ON
%
% # measure the current
% puts $s "idcaverage stat"
% set resp [gets $s]
Slot,Port 2,1 READY 140.25 mA
%
% #measure the voltage
% puts $s "vdcaverage stat"
% set resp [gets $s]
Slot,Port 2,1 READY 48.90 volts
%
% # increase the load
% puts $s "iload 2,1 i 200"
% set resp [gets $s]
%
% # measure the current
% puts $s "idcaverage stat"
% set resp [gets $s]
Slot,Port 2,1 READY 200.00 mA
%
% # measure the current, return only the value (mA), with none of the surrounding text
% puts $s "lindex \[idcaverage stat\] 3"
% set resp [gets $s]
200.00
%
% # increase the current to a point that will cause the PSE to shutdown
% puts $s "iload 2,1 i 450"
% set resp [gets $s]
%
```

```
% # measure the port voltage
% puts $s "vdcaverage stat"
% set resp [gets $s]
Slot,Port 2,1 READY 0.55 volts
%
% # measure the current
% puts $s "idcaverage stat"
% set resp [gets $s]
Slot,Port 2,1 READY 0.25 mA
%
% # perform an orderly shutdown of the port
% puts $s "psa_disconnect"
% set resp [gets $s]
% # this is a command that will cause an error
% puts $s "alt 4,2 ?"
% set resp [gets $s]
COMMAND ERROR: slot_port_check Error: Slot Number is Out of Range.  Available Slots = 1 2 3.
```

Listing 3

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Example server session, captured from a running PowerShell window.
# Note: the Tcl procs listed in Figure 1 have already been sourced into the PowerShell widow.  The
# Echo_Server statement configures the socket server, and the vwait causes the shell to enter an
# infinite loop waiting for activity on the socket.
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PSA-1,1>Echo_Server 6900
sock308
PSA-1,1>vwait forever
Accept sock392 from 127.0.0.1 port 2305
processing cmd: psa_config
returning: {0,0 PSA3000_Controller_01 3.00} {1,1 PSA3102_Test_Blade_01 3.01} {1,2
PSA3102_Test_Blade_01 3.01} {2,1 PSA3102_Test_Blade_01 3.01} {2,2 PSA3102_Test_Blade_01 3.01}
{3,1 PSA3102_Test_Blade_01 3.01} {3,2 PSA3102_Test_Blade_01 3.01} {4,1 Empty  .  } {4,2 Empty  .  }
{5,1 Empty  .  } {5,2 Empty  .  } {6,1 Empty  .  } {6,2 Empty  .  } {7,1 Empty  .  } {7,2 Empty  .  } {8,1 Empty  .  }
{8,2Empty  .  } {9,1 Empty  .  } {9,2 Empty  .  } {10,1 Empty  .  } {10,2 Empty  .  } {11,1 Empty  .  }
{11,2 Empty  .  } {12,1 Empty  .  } {12,2 Empty  .  }
processing cmd: power_port 2,1 c 0
returning: POWERED 48.90 140
processing cmd: pstatus 2,1
returning: Detection: ON
Powered: ON
Arming: OFF
Aux_LED: ON
processing cmd: idcaverage stat
```

```
returning: Slot,Port 2,1
  READY
  140.25 mA
processing cmd: vdcaverage stat
returning: Slot,Port 2,1
  READY
  48.90 volts
processing cmd: iload 2,1 i 200
returning:
processing cmd: idcaverage stat
returning: Slot,Port 2,1
  READY
  200.00 mA
processing cmd: lindex [idcaverage stat] 3
returning: 200.00
processing cmd: iload 2,1 i 450
returning:
processing cmd: vdcaverage stat
returning: Slot,Port 2,1
  READY
  0.55 volts
processing cmd: idcaverage stat
returning: Slot,Port 2,1
  READY
  0.25 mA
processing cmd: psa_disconnect
returning:
processing cmd: alt 4,2 ?
COMMAND ERROR: slot_port_check Error: Slot Number is Out of Range.  Available Slots = 1 2 3.
```

Listing 4

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Example client session, running in a generic Tcl window, connected to a server that is running the
# script "socket_server_delimited.txt".
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# this command will set the active load, and the Tcl result is an empty string.
% puts $s "iload 1,1 i 0"
% set resp [gets $s]
COMMAND SUCCESS - result: EMPTY_STRING
# this command will read the active load setting, which is returned in a non-empty string.
% puts $s "iload 1,1 ?"
% set resp [gets $s]
COMMAND SUCCESS - result: Slot,Port;1,1;Current_Load:;0.00;mA;Transition_Current:;0.00;mA;
```

Method 2 Remote Control of PowerShell Tcl Using Mailbox Files

Passing Commands via psa_command.txt

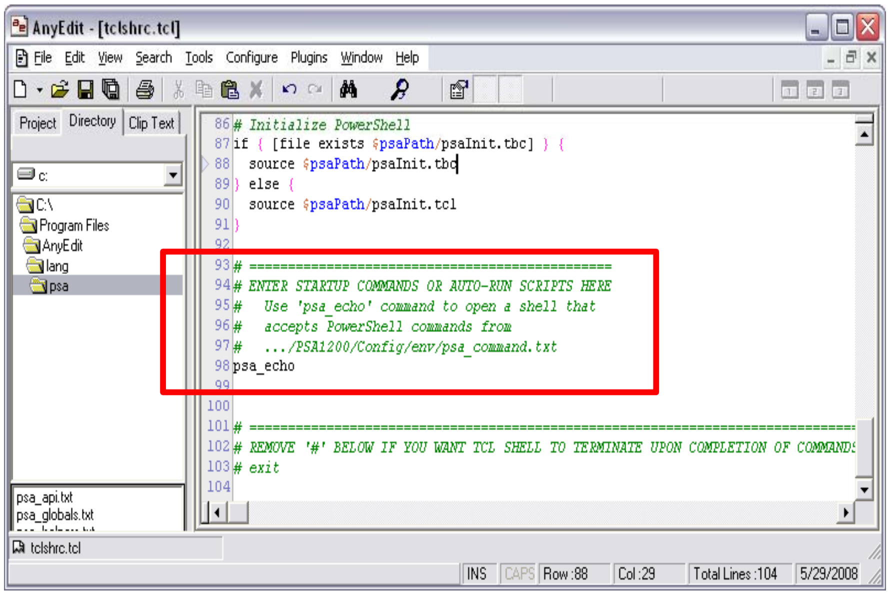
PowerShell Tcl can be opened in a manner where it will accept PowerShell (and Tcl) commands passed through a short ascii text file. The command passing file is:

c:\Program Files\Sifos\PSA1200\Config\env\psa_command.txt (*Microsoft Windows*)
$Home/Sifos/PSA1200/Config/env/psa_command.txt (*Linux/Unix*)

Any outside program that can produce a text file with PowerShell commands and arguments organized as list elements can transmit those commands to PowerShell Tcl for execution. This file will be volatile in that as soon as it is created by the outside program, it will rapidly be read and deleted by PowerShell Tcl.

NOTE: it is strongly advised that the command file be created using a temporary name (e.g. **psa_command.tmp**), then simply renamed to **psa_command.txt** after being closed, in order to avoid a race condition between PowerShell and the application that produces the command file.

To place PowerShell into the Remote Command Processing mode, a simple modification is made to the PowerShell Tcl resource file **tclshrc.tcl**. Near the bottom of that file, the commented '**psa_echo**' command is un-commented (by removing the Tcl comment character '#'), so that it will be executed when PowerShell Tcl is invoked. This will cause PowerShell Tcl to open directly into the command polling (or Remote Command Processing) mode.

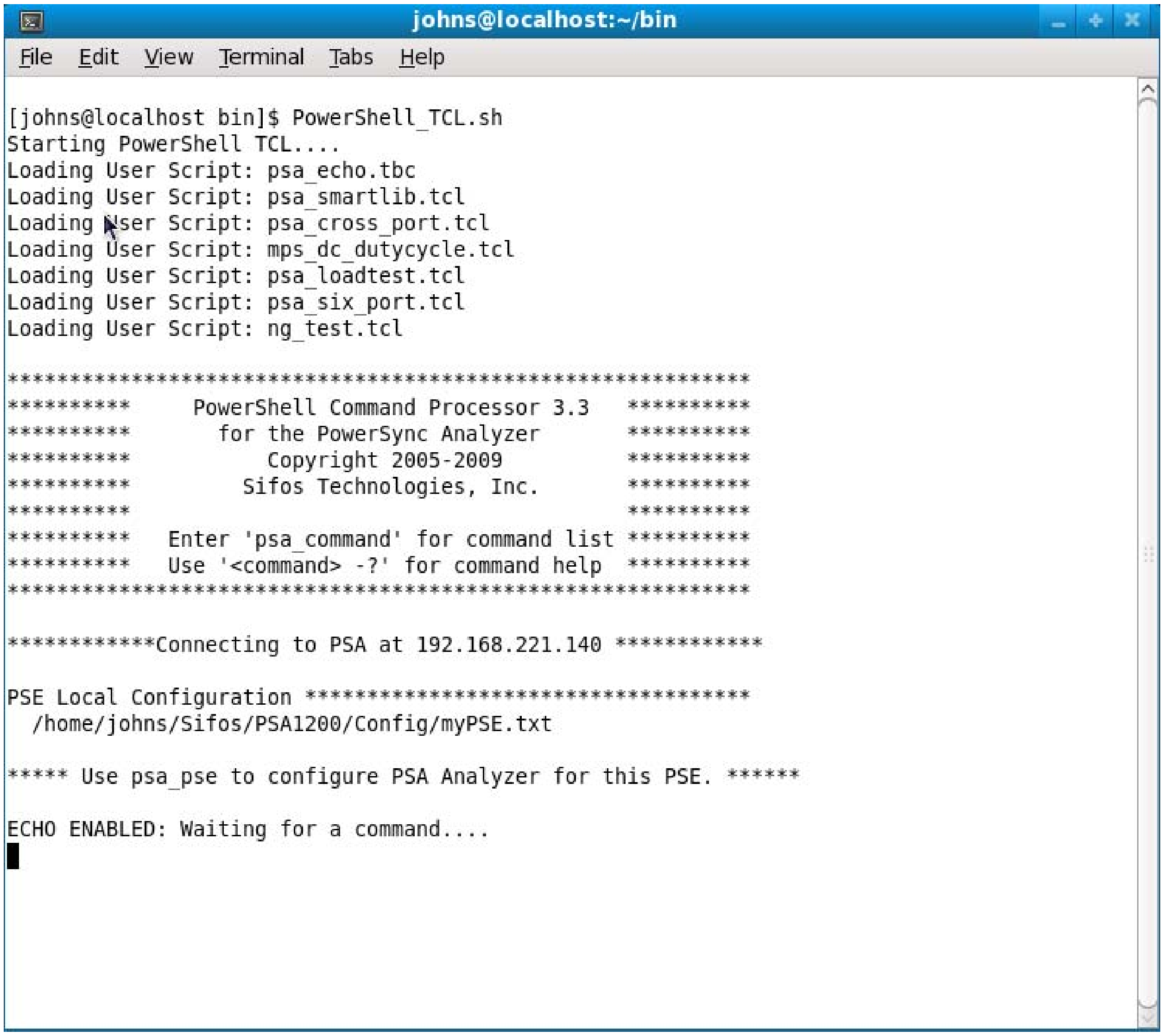

AnyEdit - [tclshrc.tcl]
File Edit View Search Tools Configure Plugins Window Help
Project Directory Clip Text
c:
C:\
Program Files
AnyEdit
lang
psa
psa_api.txt
psa_globals.txt
tclshrc.tcl
INS CAPS Row :88 Col :29 Total Lines :104 5/29/2008

```
# Initialize PowerShell
if { [file exists $psaPath/psaInit.tbc] } {
  source $psaPath/psaInit.tbc
} else {
  source $psaPath/psaInit.tcl
}

# ==============================================
# ENTER STARTUP COMMANDS OR AUTO-RUN SCRIPTS HERE
#   Use 'psa_echo' command to open a shell that
#   accepts PowerShell commands from
#   .../PSA1200/Config/env/psa_command.txt
psa_echo


# ================================================================================
# REMOVE '#' BELOW IF YOU WANT TCL SHELL TO TERMINATE UPON COMPLETION OF COMMANDS
# exit
```

Figure 4 tclshrc.tcl Modified to Enable Remote Command Processing

NOTE: **tclshrc.tcl** can be configured to bypass the initial connection dialog. The topic *Startup Considerations* below discusses methods for bypassing the initial connection dialog and delay.

NOTE: The Remote Command Processing mode may also be enabled for PowerShell Wish by modifying the Wish shell resource file **wishrc.tcl**.

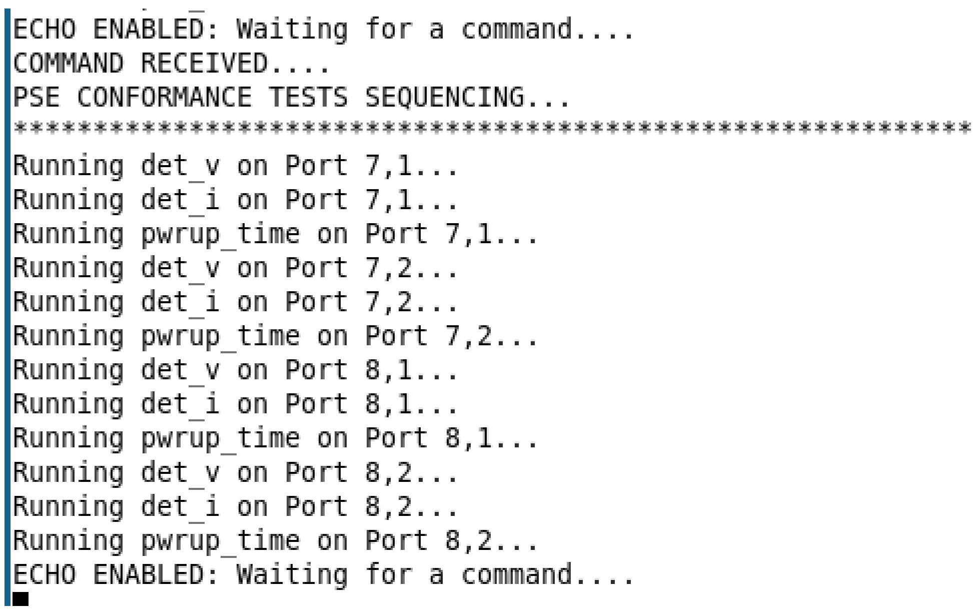


```
[johns@localhost bin]$ PowerShell_TCL.sh
Starting PowerShell TCL....
Loading User Script: psa_echo.tbc
Loading User Script: psa_smartlib.tcl
Loading User Script: psa_cross_port.tcl
Loading User Script: mps_dc_dutycycle.tcl
Loading User Script: psa_loadtest.tcl
Loading User Script: psa_six_port.tcl
Loading User Script: ng_test.tcl

************************************************************
**********     PowerShell Command Processor 3.3   **********
**********       for the PowerSync Analyzer       **********
**********         Copyright 2005-2009            **********
**********        Sifos Technologies, Inc.        **********
**********                                        **********
**********   Enter 'psa_command' for command list **********
**********   Use '<command> -?' for command help  **********
************************************************************

************Connecting to PSA at 192.168.221.140 ************

PSE Local Configuration ************************************
  /home/johns/Sifos/PSA1200/Config/myPSE.txt

***** Use psa_pse to configure PSA Analyzer for this PSE. ******

ECHO ENABLED: Waiting for a command....
```

Figure 5 PowerShell Opened for Mailbox File Control

The command passing file must be created with each PowerShell command or command sequence and the associated arguments organized as a list. Each command **must** be surrounded by braces { } in order to accomplish this. The following commands form a sequence that connects to a PSA chassis, configures it for a particular PSE type, then uses the PowerShell sequencer to run list of tests on a list of 4 ports:

```
{psa 192.168.221.140}
{psa_pse myPSE}
{set portList "7,1 7,2 8,1 8,2"}
{set testList "det_v  det_i  pwrup_time"}
{sequence -p $portList –t $testList -f}
```



```
ECHO ENABLED: Waiting for a command....
COMMAND RECEIVED....
PSE CONFORMANCE TESTS SEQUENCING...
************************************************************
Running det_v on Port 7,1...
Running det_i on Port 7,1...
Running pwrup_time on Port 7,1...
Running det_v on Port 7,2...
Running det_i on Port 7,2...
Running pwrup_time on Port 7,2...
Running det_v on Port 8,1...
Running det_i on Port 8,1...
Running pwrup_time on Port 8,1...
Running det_v on Port 8,2...
Running det_i on Port 8,2...
Running pwrup_time on Port 8,2...
ECHO ENABLED: Waiting for a command....
```

Figure 6 Shell Output While Executing Commands Received via psa_command.txt

Response Handshake via psa_response.txt

When PowerShell Tcl completes execution of all of the commands passed via psa_command.txt, it will create a short ascii text file:

c:\Program Files\Sifos\PSA1200\Config\env\psa_response.txt

This response file will simply contain the string “DONE”, and serves solely as a handshake to the outside program that command execution has completed, and that PowerShell Tcl is ready for the next set of commands. The outside program should remove the response file before submitting the next set of commands via psa_command.txt, so that the completion of command processing will be detectable.

A special command can be passed to PowerShell Tcl to end Remote Command Processing and allow it to return a normal PowerShell interactive prompt. This command is:

psa_echo_off

This command will enable any subsequent commands entered into the PowerShell Tcl resource file to execute, such as a possible "**exit**" command discussed below.

Returning Results to Client Applications

Generally, the Mailbox File method for interactive interfacing with PowerShell is best applied when the commands passed to PowerShell are actually automated test scripts written with the PowerShell API and stored in the **…PSA1200/Contrib/** directory. These scripts should be written to collect any test data or query results, and store those to any user-specified ASCII file for passing back to a host application outside PowerShell and Tcl/Tk. For example, define the PowerShell commands that you want to execute in a Tcl proc named "myTestProgram", and save that proc in the file "myTestProgram.tcl". The proc accepts a command line argument "-file", which allows you to specify where the command output is to be written. You would execute this script by passing the following commands via psa_command.txt:

```
{psa 192.168.221.106}
{source myTestProgram.tcl}
{myTestProgram  "1,1 1,2 2,1 2,2 3,1 3,2" –file myResults.txt}
```

The client application could then open "myResults.txt" and process the test results once command execution has completed.

PowerShell test sequencers for the **PSE Conformance Test Suite** and **PSE Multiport Test Suite** provide this capability through simple command arguments that allow the user to specify test result file names and locations. Those files can be either ascii text files or .csv files (.csv is subsequently processed with a template Microsoft Excel spreadsheet to produces a .xls file) that are created as those sequencers run. Sections 4.13 and 4.14 of the ***PowerSync Analyzer PSA-3000 Technical Reference Manual*** provide further information concerning the use of the **sequence** and **multiport** commands, including file handling arguments.

Terminating PowerShell Upon Script Completion

PowerShell Tcl can automatically terminate upon completion of either a Pre-Defined Batch Process or a Remote Command Processing session. This feature is also enabled by a simple edit to the PowerShell Tcl resource file:

c:\Program Files\Sifos\PSA1200\tclshrc.tcl

At the very bottom of the resource file is a header followed by the command **exit**, which is commented out. By removing the Tcl comment character '#', the **exit** command will become the final command executed in PowerShell, and the PowerShell console will close automatically.

When the Remote Command Processing mode is invoked via the **psa_echo** command, the command **psa_echo_off** must be passed to PowerShell Tcl to terminate Remote Command Processing mode. Once Remote Command Processing mode is terminated, the **tclshrc.tcl** script will complete (all statements below **psa_echo**), which will now include the **exit** command, terminating PowerShell.

NOTE: even though it is possible to pass individual commands to PowerShell using this method, it is NOT intended for use as the interface to the command level API. It was implemented primarily to allow remote execution of complete scripts, which produce reports as opposed to output that the controlling program needs to make decisions with. If you want to pass individual commands to PowerShell, and receive the output from those commands, the Socket Interface method should be used.

Startup Considerations

Whenever **PowerShell Tcl** is initiated, the user is presented with a command prompt to either re-connect to the most recently connected PSA or to enter a new PSA address and establish a different connection. By default, this command prompt will time out after about 8 seconds and PowerShell will then attempt to connect to the most recently connected PSA address or subsequently to any other known PSA addresses.

Users may elect to either bypass this prompt entirely, or to configure the time delay enforced by the connection prompt prior to connecting to a default (most recent or other known) address. This is done by setting the appropriate value of the global variable **psaConnectPause,** found near the beginning of the **tclshrc.tcl** and/or **tclshrc_psapi.tcl** PowerShell Tcl initialization files. Normally, this global variable will be set to 8 seconds. If set to zero, the initial connection prompt will be entirely bypassed and PowerShell Tcl will attempt to open a connection to the most recently connected PSA address. The delay associated with the connection prompt may be set between 2 and 60 seconds.

Whenever **PowerShell Wish** is initiated, the user is presented with a PSA Connection dialog that must, by default, be completed before the application will fully start. Users may elect to bypass this dialog entirely by setting the value of the global variable **psaConnectPause,** found near the beginning of **wishrc.tcl** and **wishrc_psapi.tcl**. If the value is set to 1 (default), the initial PSA connection dialog will always be displayed, and will wait indefinitely for a user selection. If the value is set to 0, the connection dialog will be bypassed and the application will automatically attempt to connect to the most recently connected PSA address.

NOTE: It is generally not a good idea to bypass the initial connection prompt (dialog) if multiple (2 or more) PowerSync Analyzers exist on a common LAN and are shared by multiple users. Bypassing the initial connection dialog increases the risk that two users will attempt to control the same PowerSync Analyzer at the same time (the instrument only supports a single session).

If you have 2 or more PowerSync Analyzers, you can force the connection to a specific PowerSync Analyzer by modifying the PowerShell variable **Default_PSA_Address** in the file **~Sifos/PSA1200/Config/env/psa_env.txt**, defining the IP address of the desired instrument. **NOTE:** do not modify any other values in the **~psa_env.txt** file, so you won't interfere with other PowerShell functionality. Also, even if you connect to the 'wrong' PowerSync Analyzer, you can issue a command to force a connection to the 'correct' PowerSync Analyzer at the beginning of your script(s). This is done with the **psa** command.

Method 3 Remote Control of PowerShell Tcl Using Batch Mode

PowerShell can be executed from an external application program, and will load and execute a script defined as an argument to that executable. The shell will terminate when the script execution has completed.

Invoking PowerShell Tcl

Microsoft Windows Platforms

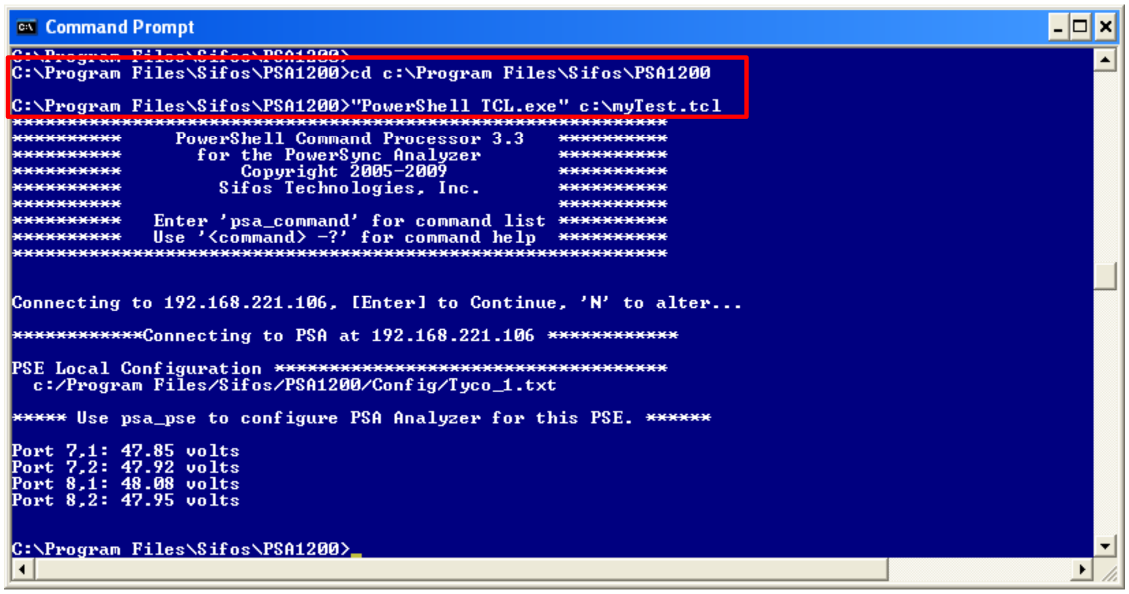
On Microsoft Windows PC's, the executable program **c:\Program Files\Sifos\PSA1200\PowerShell Tcl.exe** will launch **PowerShell Tcl**. A PowerShell Tcl Test Script can be automatically executed in that shell if specified as an argument to the **PowerShell Tcl.exe** command. Upon completion of the PowerShell script, PowerShell Tcl will close.

In the following example, a PowerShell Test script will be launched from a Windows command shell (Start -> All Programs -> Accessories -> Command Prompt). This script will measure port voltages on a PSE connected to ports 7,1 – 8,2 of the PowerSync Analyzer located at IP address 192.168.221.106. Prior to launching this command, the **tclshrc.tcl** resource file discussed above in *Startup Considerations* should be modified to bypass the user connection dialog.

Windows Command Shell Commands	PowerShell Test Script named c:\myTest.tcl

`cd c:\Program Files\Sifos\PSA1200`
`"PowerShell Tcl.exe" c:\myTest.tcl`
	`psa 192.168.221.106`
	`alt 99,99 A`
	`polarity 99,99 neg`
	`foreach port "7,1 7,2 8,1 8,2" {`
	`  set status [power_port $port c 0]`
	`  append Vport "Port $port: [lindex $status 1] volts\n"`
	`  psa_disconnect $port`
	`}`
	`puts $Vport`

The above example produces the following result:

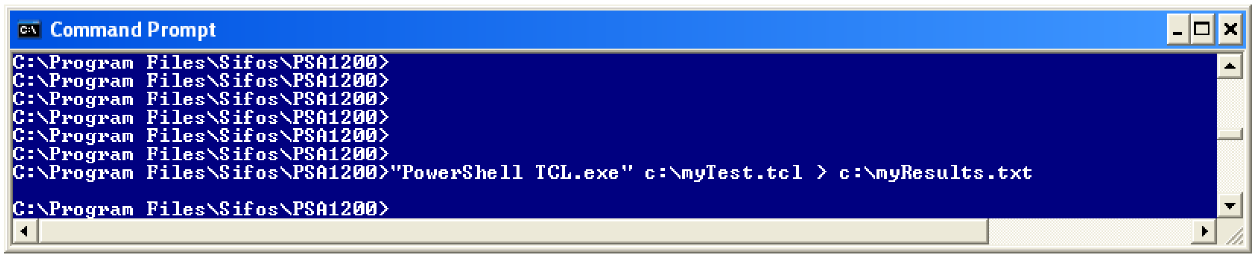


```
C:\Program Files\Sifos\PSA1200>cd c:\Program Files\Sifos\PSA1200

C:\Program Files\Sifos\PSA1200>"PowerShell TCL.exe" c:\myTest.tcl
************************************************************
***********     PowerShell Command Processor 3.3    ***********
***********       for the PowerSync Analyzer        ***********
***********          Copyright 2005-2009            ***********
***********        Sifos Technologies, Inc.         ***********
***********                                         ***********
***********   Enter 'psa_command' for command list  ***********
***********   Use '<command> -?' for command help   ***********
************************************************************


Connecting to 192.168.221.106, [Enter] to Continue, 'N' to alter...

************Connecting to PSA at 192.168.221.106 ************

PSE Local Configuration ************************************
  c:/Program Files/Sifos/PSA1200/Config/Tyco_1.txt

***** Use psa_pse to configure PSA Analyzer for this PSE. ******

Port 7,1: 47.85 volts
Port 7,2: 47.92 volts
Port 8,1: 48.08 volts
Port 8,2: 47.95 volts

C:\Program Files\Sifos\PSA1200>
```

Figure 7 Remote Execution of PowerShell Test Script myTest.tcl

The test results from myTest.tcl can be readily redirected to a text file for reading by an outside application. This is done by simply modifying the Shell Command above to direct output to a specified file such as c:\myResults.txt:

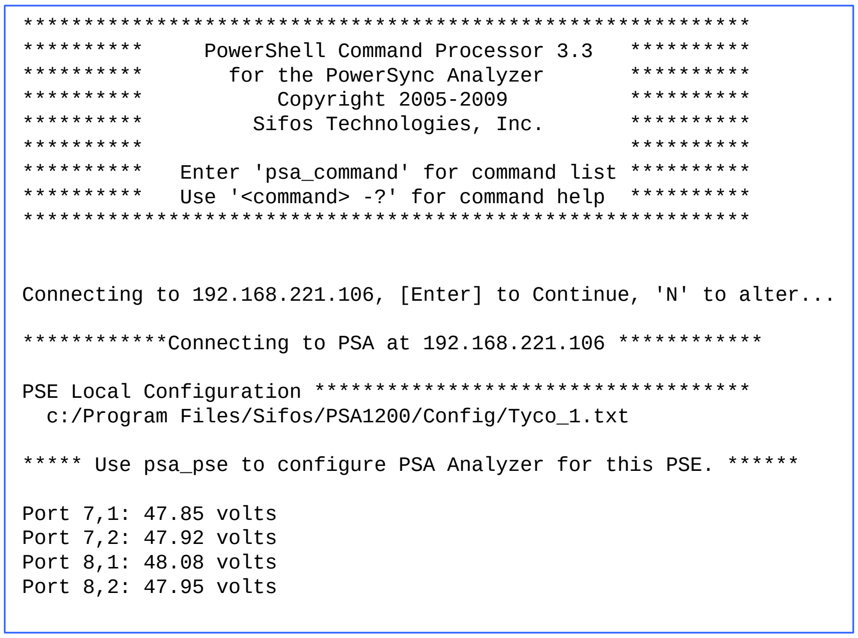
"PowerShell Tcl.exe" c:\myTest.tcl > c:\myResults.txt



```
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>
C:\Program Files\Sifos\PSA1200>"PowerShell TCL.exe" c:\myTest.tcl > c:\myResults.txt

C:\Program Files\Sifos\PSA1200>
```

Figure 8 Routing Test Output to a Text File

This produces the following text file in **c:\myResults.txt**:

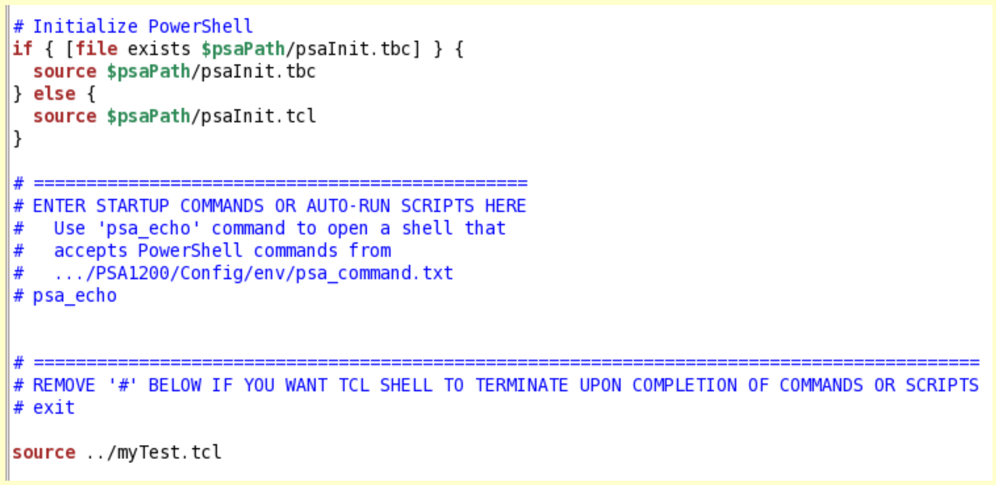


```
************************************************************
**********     PowerShell Command Processor 3.3   **********
**********       for the PowerSync Analyzer       **********
**********         Copyright 2005-2009            **********
**********        Sifos Technologies, Inc.        **********
**********                                        **********
**********   Enter 'psa_command' for command list **********
**********   Use '<command> -?' for command help  **********
************************************************************


Connecting to 192.168.221.106, [Enter] to Continue, 'N' to alter...

************Connecting to PSA at 192.168.221.106 ************

PSE Local Configuration ************************************
  c:/Program Files/Sifos/PSA1200/Config/Tyco_1.txt

***** Use psa_pse to configure PSA Analyzer for this PSE. ******

Port 7,1: 47.85 volts
Port 7,2: 47.92 volts
Port 8,1: 48.08 volts
Port 8,2: 47.95 volts
```

Figure 9 Text File Output from myTest.tcl

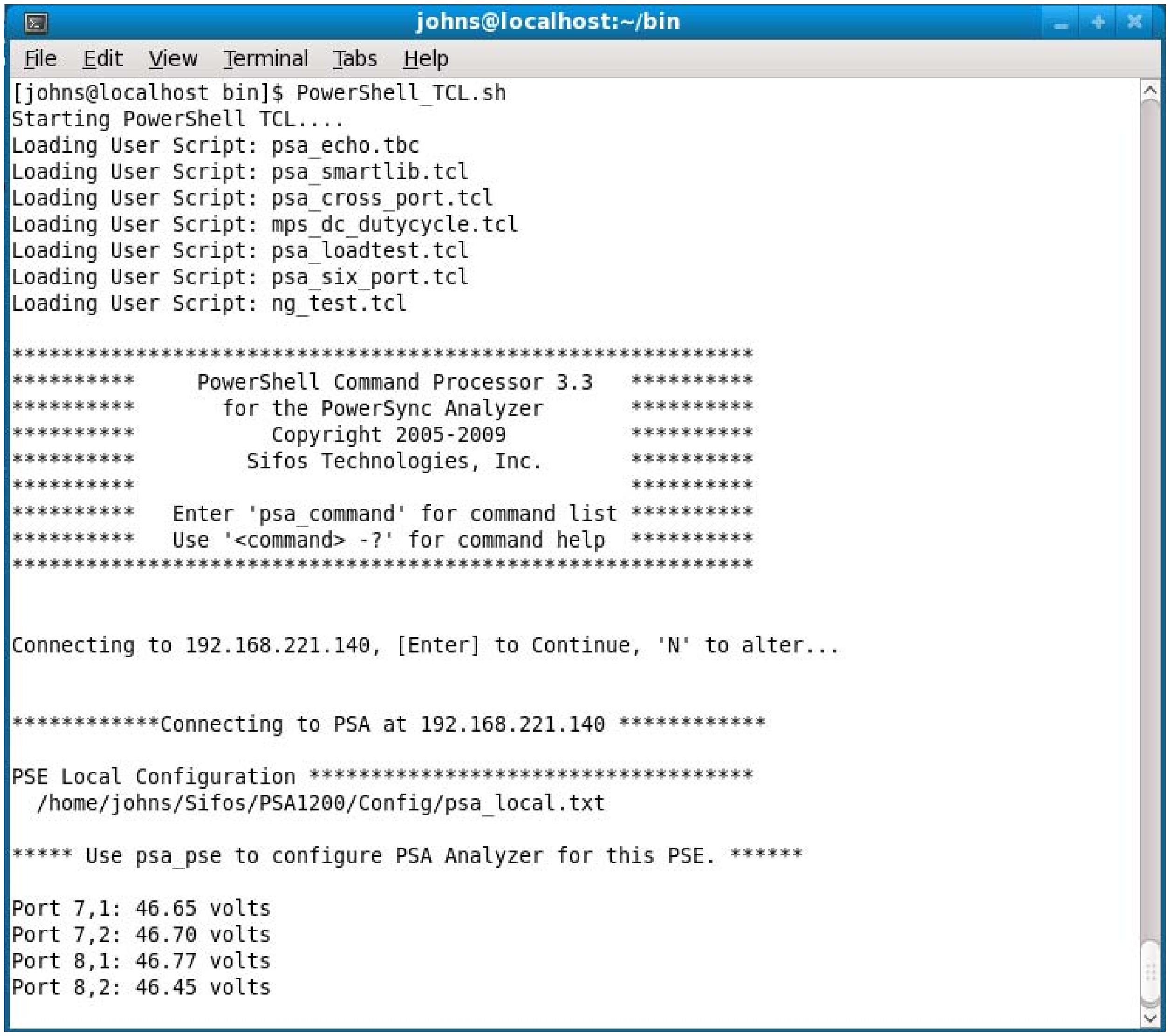
Other Platforms

A more general way to accomplish this same task, which works with both Microsoft Windows and other operating systems, is to execute the binary for Tcl with an argument to source the PowerShell Tcl resource file, **tclshrc.tcl** (or **tclshrc_psapi.tcl**). This will require an additional edit to add the user application script at the very end of **tclshrc.tcl** (or **tclshrc_psapi.tcl**). This is illustrated in the following example with a similar **myTest.tcl** user script (in this case, modified to connect to the chassis at 192.168.221.140).

Windows Command Shell Command	**cd c:\Program Files\Tcl\bin** **tclsh84.exe "c:\Program Files\Sifos\PSA1200\tclshrc.tcl"**
Linux Shell Command	**cd $HOME/bin** **PowerShell_TCL.sh** **NOTE: this script copies $HOME/Sifos/PSA1200/tclshrc.tcl to $HOME/.tclshrc.tcl** **The file $HOME/.tclshrc.tcl will be deleted when the shell is terminated.** **Any edit needs to be performed on $HOME/Sifos/PSA1200/tclshrc.tcl.**
tclshrc.tcl Edits	

```
# Initialize PowerShell
if { [file exists $psaPath/psaInit.tbc] } {
  source $psaPath/psaInit.tbc
} else {
  source $psaPath/psaInit.tcl
}

# ===============================================
# ENTER STARTUP COMMANDS OR AUTO-RUN SCRIPTS HERE
#   Use 'psa_echo' command to open a shell that
#   accepts PowerShell commands from
#   .../PSA1200/Config/env/psa_command.txt
# psa_echo


# ==========================================================================================
# REMOVE '#' BELOW IF YOU WANT TCL SHELL TO TERMINATE UPON COMPLETION OF COMMANDS OR SCRIPTS
# exit

source ../myTest.tcl
```

The above example produces the following result, when executed on fedora-10-i386:



```
[johns@localhost bin]$ PowerShell_TCL.sh
Starting PowerShell TCL....
Loading User Script: psa_echo.tbc
Loading User Script: psa_smartlib.tcl
Loading User Script: psa_cross_port.tcl
Loading User Script: mps_dc_dutycycle.tcl
Loading User Script: psa_loadtest.tcl
Loading User Script: psa_six_port.tcl
Loading User Script: ng_test.tcl

************************************************************
**********     PowerShell Command Processor 3.3   **********
**********       for the PowerSync Analyzer       **********
**********           Copyright 2005-2009          **********
**********         Sifos Technologies, Inc.       **********
**********                                        **********
**********    Enter 'psa_command' for command list **********
**********    Use '<command> -?' for command help  **********
************************************************************


Connecting to 192.168.221.140, [Enter] to Continue, 'N' to alter...


************Connecting to PSA at 192.168.221.140 ************

PSE Local Configuration ************************************
  /home/johns/Sifos/PSA1200/Config/psa_local.txt

***** Use psa_pse to configure PSA Analyzer for this PSE. ******

Port 7,1: 46.65 volts
Port 7,2: 46.70 volts
Port 8,1: 46.77 volts
Port 8,2: 46.45 volts
```

Figure 10 Remote PowerShell Test Script Executed Via tclsh8.4 Execution

File Permissions

Users should be aware that when PSA software is installed the PowerShell initialization files **tclshrc.tcl**, **tclshrc_psapi.tcl**, **wishrc.tcl**, and **wishrc_psapi.tcl** are all read-only files. To modify them, their permissions must be changed. Also, when software upgrades are installed, users should assume that any modifications made locally to those files will be lost, so it is **important** that copies of those modified files are maintained in other directories.