Sifos Technologies

PhyView® Analyzer

PVA-3102 Test Blade in PSA-3000 or PVA-3002

PhyView API Library Reference Manual

Version 4.1.8

Revised March 19, 2018

Sifos Technologies, Inc.
(978) 640-4900 Phone
(978) 640-4990 FAX

Disclaimer

Table of Contents

1. Introduction

1.1. PhyView API Library Introduction

The Sifos Technologies PhyView Analyzer is a test instrument designed to be controlled from a PC over a TCP/IP network connection (10/100BaseT). The software environment used to control a PhyView Analyzer is **PowerShell PSA**, a powerful, interpretive scripting language built on the widespread Tcl language. From an automation perspective, the native Application Programming Interface (API) for PowerShell PSA is a set of Tcl commands, which require a Tcl interpreter to execute.

To facilitate automation control of the PhyView Analyzer from alternative languages, a binary API Library (referred to below as "the API Library") has been created. The API library programmatically initializes a Tcl interpreter, translates the information passed by the caller for each function into the related PowerShell PSA command, evaluates that PowerShell PSA command with the Tcl interpreter, and returns the result of that command evaluation to the caller. Calls will not return until the related Tcl command execution has been completed, so no special pacing should be required in the calling context.

The ***PhyView API Library Reference*** document defines the functions that are specifically related to the PhyView Analyzer. This document does not describe the theory of operation of a PhyView Analyzer, and does not include the full description of the actual PowerShell PSA Tcl commands that each API function encapsulates. This API manual should be used in conjunction with the ***PhyView Analyzer PVA-3000 Reference Manual***.

The PowerShell API Library is published as a Dynamic Link Library (.dll) for use on Microsoft Windows platforms, and as a Shared Library (.so) for use on Linux platforms.

On a Windows platform, any language capable of calling Win32 API functions should be able to use the PowerShell API library. On a Linux platform, any application capable of calling C functions located in a shared library should be able to use the PowerShell API.

The API library is considered a "contributed" utility, and is provided for the convenience of end users who are not able to utilize Tcl directly, and choose not to employ one of the other remote control methods available for the PowerSync Analyzer (described in the Application Note ***Remote Control of PowerShell Tcl***). The API library has been tested against the release of PowerShell PSA noted below. Given the multitude of programming environments available, Sifos can only provide limited support with regard to the use of the API library with non-standard environments.

1.2. System Requirements

For Microsoft Windows:

Windows NT4.0 SP4 or later, through Windows 7
PSA Software version 4.0.6 or later
Tcl/Tk version 8.4.5 or later versions of 8.4 (the PSA Software has currently been qualified up to version 8.4.20)
Note: Tcl 8.5 and above are not supported.
PowerShell API Library (PowerShellAPI.dll) version 4.1.1 or later

For Linux:

tested on SuSE version 10.2; OpenSuSE 11; Fedora 2.6.9 (kernel 2.6, GLIBC_2.0, GLIBC_2.1.3); Debian 6.2
PSA Software version 4.0.4 or later
Tcl/Tk version 8.4.5 or later versions of 8.4
Note: Tcl 8.5 and above are not supported.
PowerShell API Library (libPowerShellAPI.so) version 4.1.1 or later. (Library requires libc.so.6, libm.so.6, libdl.so.2.)

Test Blades

Sifos PVA-3102

1.3. Installation

The API library binary file must be placed in a location that the application program you are using can find at runtime. For Windows platforms, there are specific rules that the operating system uses to locate a DLL:

searches under the current working directory
searches under C:\Windows
searches under C:\Windows\System32
searches under directories defined in the PATH environment variable

The .h, .lib, and .bas files should be placed wherever necessary to access them from your application development environment. NOTE: the .lib file was produced with a Microsoft C version 6 linker.

For Linux platforms, the .so file should be placed under /usr/lib (or an alternative directory as defined by a local administrative policy, if necessary).

The .h file should be placed wherever necessary to access it from your application development environment.

1.4. Use with a PSA Chassis Containing PSA/PSL-3102 Test Blades

The API library also contains support for the Sifos PowerSync Analyzer (PSA-3102 Test Blades) and PowerSync Programmable Load (PSL-3102 Test Blades). There are a small number of functions that are common to both PowerSync Analyzer and PhyView Analyzer capabilities (listed below). All other functions defined in this document are specific to the PVA-3102 Test Blade type. The PowerSync Analyzer related functions are defined in a separate document, the ***PowerShell API Library Reference Manual*.**

Commands that are common to both Test Blade types:

PowerShell_ManageStdout
PowerShell_Init
PowerShell_UnInit
PowerShell_ConnectToChassis
PowerShell_GetInitStatus
PowerShell_GetErrString
PowerShell_Inventory
PowerShell_GetVersion
PowerShell_ProcessCommand
PowerShell_SetConfigTrigPort
PowerShell_GetConfigTrigPort
PowerShell_TrigOut

1.5. Use in a Multi-threaded Program

The API library functions are **not** thread safe. Due to the architecture of the PhyView Analyzer instrument, which only supports a single connection at any instant in time, the underlying Tcl extension environment (PowerShell PSA) that is used to control the instrument only supports a single thread use model. As a result, there has been no engineering effort devoted to making the API functions thread safe.

It is incumbent on the end user to use the API library in a manner where functions are called by a single thread at any one point in time. Each function call is “atomic”, where the function call marshals arguments, submits the associated PowerShell PSA command to the encapsulated Tcl shell for evaluation, and does not return until the PowerShell PSA command execution has completed. Multiple threads need to use some form of access control, such as semaphores or mutex’s, to insure that one and only one thread is performing an API function call at any single point in time. The semaphore or mutex **must not** be released by the thread that owns the semaphore or mutex until the API library function call has returned.

If more than one thread calls API functions simultaneously, unpredictable results will occur, with a high probability that one of the threads will encounter a socket error at the lowest level of PowerShell PSA.

1.6. Reference Manual Organization

Section 2 contains the PhyView API function definitions, and related enumeration type definitions for any language able to call a standard C function.

Section 3 contains information about using the PhyView API library with Visual Basic 6 and Visual Basic .NET languages.

Section 4 contains information about using the PhyView API library with LabView.

Section 5 contains example code showing how to use the PhyView API library.

2. PhyView API Defintions

The PhyView API function prototypes and associated enumerations are defined in the file PhyViewAPI.h

2.1. Calling Convention

On Microsoft Windows platforms, the calling convention used by the API functions is __stdcall.

There is no explicit definition of the calling convention for Linux platforms. The default calling convention used is defined by gcc (specifically, gcc4.2).

2.2. Error Handling

Each of the PowerShell API functions returns a status: TCL_OK | TCL_ERROR

The application program should **always** test the status returned by an API function.

When an API function returns the TCL_ERROR status, the calling application should immediately call the API function that returns a message associated with that error. That function is PowerShell_GetErrString. The error text buffer managed by the library will be overwritten by the next API function call, so the error message will be 'lost'.

2.3. Enumerations

The enumerations used by the various API functions are defined in the include file PhyViewAPI.h. If the language being used does not support enumerations, pass the integer value defined below.

```
enum _ePVAConnect { PVA_CONNECT_UNKNOWN = 0, PVA_PHY = 1, PVA_THROUGH = 2 };
typedef enum _ePVAConnect ePVAConnect;

enum _ePVAImpair { PVA_IMPAIR_UNKNOWN = 0, PVA_NORMAL = 1, PVA_IMPAIR = 2 };
typedef enum _ePVAImpair ePVAImpair;

enum _eNoiseState { PVA_NOISE_UNKNOWN = 0, PVA_NOISE_OFF = 1, PVA_NOISE_ON = 2 };
typedef enum _eNoiseState eNoiseState;

enum _ePVATxClkState { PVA_CLK_UNKNOWN = 0, PVA_CLK_NORMAL = 1, PVA_CLK_OFFSET = 2,
        PVA_CLK_JITTER = 3 };
typedef enum _ePVATxClkState ePVATxClkState;

enum _ePVABoolean { PVA_FALSE = 0, PVA_TRUE = 1 };
typedef enum _ePVABoolean ePVABoolean;

enum _ePVASpeed { PVA_SPEED_UNKNOWN = 0, PVA_10 = 1, PVA_100 = 2, PVA_1000 = 4, PVA_10_100 = 8,
        PVA_SPEED_AUTO = 16, PVA_SPEED_UNDETERMINED = 32 };
typedef enum _ePVASpeed    ePVASpeed;

enum _ePVADuplex { PVA_DUPLEX_UNKNOWN = 0, PVA_AUTO_DUPLEX = 1, PVA_FULL_DUPLEX = 2,
        PVA_HALF_DUPLEX = 3, PVA_VARIOUS_DUPLEX = 4 };
typedef enum _ePVADuplex ePVADuplex;

enum _ePVAPolarity { PVA_POL_UNKNOWN = 0, PVA_POL_AUTO = 1, PVA_MDI = 2, PVA_MDIX = 3 };
```

```
typedef enum _ePVAPolarity ePVAPolarity;

enum _ePVAGigMode { PVA_GIG_UNKNOWN = 0,PVA_GIG_AUTO = 1, PVA_GIG_MASTER = 2,
        PVA_GIG_SLAVE = 3, PVA_GIG_FAULT = 4, PVA_GIG_NA = 5 };
typedef enum _ePVAGigMode ePVAGigMode;

enum _ePVALinkState { PVA_LINKSTATE_UNKNOWN = 0, PVA_LINKED = 1, PVA_DOWN = 2,
        PVA_UNLINKED = 3, PVA_UNSTABLE = 4, PVA_DISCONNECTED = 5 };
typedef enum _ePVALinkState ePVALinkState;

enum _ePVAMeasTypes { PVA_MEAS_UNKNOWN = 0, PVA_MEAS_PSD = 1, PVA_MEAS_ECHO = 2,
        PVA_MEAS_XTALK = 4 };
typedef enum _ePVAMeasTypes ePVAMeasTypes;

enum _ePVAPair { PVA_PAIR_UNKNOWN = 0, PVA_PAIR1 = 1, PVA_PAIR2 = 2, PVA_PAIR3 = 3,
        PVA_PAIR4 = 4 };
typedef enum _ePVAPair ePVAPair;

enum _ePVATrig { PVA_TRIG_UNKNOWN = 0, PVA_TRIG_OFF = 1, PVA_TRIG_EXTERNAL = 2 };
typedef enum _ePVATrig ePVATrig;

enum _ePVATimeout { PVA_TIMEOUT_UNKNOWN = 0, PVA_TIMEOUT_10S = 1, PVA_TIMEOUT_100S = 2 };
typedef enum _ePVATimeout ePVATimeout;

enum _ePVAMeterStatus { PVA_METER_UNKNOWN = 0, PVA_METER_UNLINKED = 1,
        PVA_METER_READY = 2, PVA_METER_ARMED = 3, PVA_METER_MEASURING = 4,
        PVA_METER_TIMEOUT = 5, PVA_METER_LINK_DROP = 6, PVA_METER_DISCONNECTED = 7 };
typedef enum _ePVAMeterStatus ePVAMeterStatus;

enum _ePVAPairGroup { PVA_PAIRS_UNKNOWN = 0, PVA_PAIRS12 = 1, PVA_PAIRS13 = 2,
        PVA_PAIRS14 = 3, PVA_PAIRS23 = 4, PVA_PAIRS24 = 5, PVA_PAIRS34 = 6 };
typedef enum _ePVAPairGroup ePVAPairGroup;

enum _ePVASample { PVA_SAMPLE_UNKNOWN = 0, PVA_LINK = 1, PVA_REMOTE = 2, PVA_LOCAL = 3 };
typedef enum _ePVASample ePVASample;

enum _ePVAPeriod { PVA_PERIOD_UNKNOWN = 0, PVA_20MSEC = 1, PVA_50MSEC = 2,
        PVA_100MSEC = 3 };
typedef enum _ePVAPeriod ePVAPeriod;

enum _ePVAPair14State { PVA_PAIR_STATE_UNKNOWN = 0, PVA_SWAPPED = 1, PVA_NOT_SWAPPED = 2,
        PVA_PAIR_STATE_NA = 3 };
typedef enum _ePVAPair14State ePVAPair14State;

enum _ePVAFrameCount { PVA_FRAME_COUNT_UNKNOWN = 0, PVA_CONTINUOUS = 1, PVA_32K = 2,
        PVA_128K = 3, PVA_512K = 4, PVA_1024K = 5 };
typedef enum _ePVAFrameCount ePVAFrameCount;

enum _ePVATxRate { PVA_TX_RATE_UNKNOWN = 0, PVA_LINE = 1, PVA_MED = 2, PVA_SLOW = 3 };
typedef enum _ePVATxRate ePVATxRate;

enum _ePVATxStatus { PVA_TX_STATUS_UNKNOWN = 0, PVA_TX_IDLE = 1, PVA_TX_ACTIVE_CONT = 2,
        PVA_TX_ACTIVE_BURST = 3, PVA_TX_UNLINKED = 4, PVA_TX_DISCONNECTED = 5 };
typedef enum _ePVATxStatus ePVATxStatus;
```

```
enum _ePVARxStatus { PVA_RX_STATUS_UNKNOWN = 0, PVA_RX_IDLE = 1, PVA_RX_MEASURING = 2,
        PVA_RX_UNLINKED = 3, PVA_RX_DISCONNECTED = 4 };
typedef enum _ePVARxStatus ePVARxStatus;

enum _ePVAAutoNeg { PVA_AUTONEG_UNKNOWN = 0, PVA_AUTONEG_NO = 1,
        PVA_AUTONEG_BASE = 2, PVA_AUTONEG_EXTENDED = 3, PVA_AUTONEG_NA = 4 };
typedef enum _ePVAAutoNeg ePVAAutoNeg;

enum _ePVADuplexSupport { PVA_DUPLEX_SUPPORT_UNKNOWN = 0, PVA_FULL_DUPLEX_ONLY = 1,
        PVA_HALF_DUPLEX_ONLY = 2, PVA_HALF_plus_FULL = 3, PVA_DUPLEX_NO = 4,
        PVA_DUPLEX_NA = 5 };
typedef enum _ePVADuplexSupport ePVADuplexSupport;

enum _ePVAPause { PVA_PAUSE_UNKNOWN = 0, PVA_NO_PAUSE = 1, PVA_RESPOND = 2,
        PVA_XMIT = 3, PVA_RESPOND_plus_XMIT = 4 };
typedef enum _ePVAPause ePVAPause;

enum _ePVALinkOK { PVA_LINK_OK_UNKNOWN = 0, PVA_LINK_OK_NO = 1, PVA_LINK_OK_YES = 2,
        PVA_LINK_OK_GIG_ERR = 3, PVA_LINK_OK_NA = 4, PVA_LINK_OK_FAULT = 5 };
typedef enum _ePVALinkOK ePVALinkOK;

enum _ePVARxOK { PVA_RX_OK_UNKNOWN = 0, PVA_RX_OK_NO = 1, PVA_RX_OK_YES = 2,
        PVA_RX_OK_NA = 3 };
typedef enum _ePVARxOK ePVARxOK;

enum _ePVAReportType { PVA_REPORT_UNKNOWN = 0, PVA_SPREADSHEET_REPORT = 1,
        PVA_TEXT_REPORT = 2 };
typedef enum _ePVAReportType ePVAReportType;
```

2.4. PowerShell Functions Applicable to the PhyView Analyzer

The PowerShellAPI functions listed below provide capabilities that are common to both the PowerSync Analyzer and the PhyView Analyzer, including connecting to a chassis, obtaining an inventory, and retrieving error information from the API library. The function prototypes and associated enumerations are defined in the file PowerShellAPI.h. These functions are described here for the user's convenience.

2.4.1. PowerShell_Init

function: PowerShell_Init (char *szChassisIPAddress, char *szRCFile);

description: loads and initializes a Tcl interpeter, and evaluates the defined rc file, which will load the PowerShell environment.

NOTE: this function can only be called **once** in the lifetime of a process, when the library is initially loaded. To connect to another chassis once the library has been loaded, call the API function PowerShell_ConnectToChassis.

An error will be returned if this function is called more than once. A service function has been provided that allows the initialization state of the library to be interrogated – see PowerShell_GetInitStatus.

parameters: char *szChassisIPAddress - the IP address of the chassis to connect to, in dotted decimal format.

char *szRCFile - name of the rc file to be eval'ed by the Tcl interpreter. This must be a file name (not a path), which must exist under c:/program files/sifos/psa1200. If NULL is passed as this argument, the default file tclshrc.tcl will be eval'ed.

NOTE: the rc file used should be modified to set `psaConnectPause` to 0, to minimize the time that the initial PSA connection will take. An rc file configured with this setting is provided with the standard PowerShell PSA software release, named tclshrcAPI_LIB.tcl.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.2. PowerShell_ManageStdout

function: PowerShell_ManageStdout (ePSABoolean bCloseStdout);

description: allows caller to direct the library to manage the Tcl channels TCL_STD_OUT and TCL_STDERR, which are not available when running in some processes. For example, this situation has been encountered when using the API library with a "Windows Application" constructed with Visual Basic (both VB6 and VB.NET), and with LabView. A GUI process running on Windows will **require** this function to be called **prior** to calling PowerShell_Init. Closing stdout is not required when running in a console (text mode) process.

parameters: ePSABoolean bCloseStdout - PSA_FALSE = don't close stdout | PSA_TRUE = close stdout

returns: TCL_OK

2.4.3. PowerShell_UnInit

function: PowerShell_UnInit (void);

description: deletes the Tcl interpreter created when PowerShell_Init was called, and unloads tcl84.dll.

parameters: none

returns: TCL_OK

2.4.4. PowerShell_AllowDemoModeOperation

function: PowerShell_AllowDemoModeOperation (ePSABoolean bState, eDemoType eType, int numSlots);

description: allows the library to be configured to force the underlying PowerShell PSA Tcl session to operate in DEMO mode. This allows various function calls to be used without access to hardware.

NOTE: not all functions operate in demo mode. {*editors note mark functions that do support this*}

parameters: ePSABoolean bState - PSA_FALSE = don't allow demo mode PSA_TRUE = force demo mode

eDemoType eType – Legal values:

DEMO_TYPE_PSA3000 | DEMO_TYPE_PSL3000 | DEMO_TYPE_PSA1200 | DEMO_TYPE_PSL1200 | DEMO_TYPE_SA | DEMO_TYPE_PVA3000

int numSlots – Legal values: 1..12

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.5. PowerShell_IsDemoModeOn

function: PowerShell_IsDemoModeOn (ePSABoolean *pbState);

description: interrogates the embedded PowerShell PSA, and returns the response of "psa_demo ?".

parameters: ePSABoolean *pbState- location to store the demo mode state. Values returned:

PSA_TRUE | PSA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.6. PowerShell_GetInitStatus

function: PowerShell_GetInitStatus (eInitStatus *psStatus);

description: returns the initialization status of the library. If the PowerShell_Init function has been called, the state will be PSA_INITIALIZED.

parameters: eInitStatus *psStatus - the location to store the init state in. Values returned:

PSA_INITIALIZED | PSA_NOT_INITIALIZED

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.7. PowerShell_ConnectToChassis

function: PowerShell_ConnectToChassis (char *szChassisIPAddress);

description: connects to the specified chassis (if that chassis is powered on an accessible at the defined address).

parameters: char *szChassisIPAddress - the IP address of the chassis to connect to, in dotted decimal format.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.8. PowerShell_Inventory

function: PowerShell_Inventory (char *szInventory, int iLenInventory);

description: performs an inventory of the chassis, using the PowerShell command "psa_config". The inventory is returned as a semi-colon delimited list (there are commas in parts of the response text, which precludes formatting the string with comma delimiters).

parameters: char *szInventory - a character array that will be used to return the inventory string in.

int iLenInventory - the length of the character array passed as the szInventory parameter. If the indicated length is shorter than the length of the inventory string, the inventory string will be truncated to fit the available space.

NOTE: this parameter MUST accurately define the length of szInventory, to prevent a buffer overrun. The underlying API library is implemented in C, which does not provide any means to determine size or rank of a character array.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.9. PowerShell_GetErrString

function: PowerShell_GetErrString (char *szErrStr, int iLenErrStr);

description: retrieves an error string maintained by the API library. This string will be "No Error" if the last function call succeeded, or will contain a message related to a failure if the the last function call returned an error indication.

NOTE: this string is not persistent. It will be overwritten each time a function in the API library is called, so it should be retrieved immediately after a function indicates an error, to prevent losing access to the message associated with that error.

parameters: char *szErrStr - a character array that will be used to return the error string in.

int iLenErrStr - the length of the character array passed as the szErrStr parameter. If the indicated length is shorter than the length of the error string, the error string will be truncated to fit the available space.

NOTE: this parameter **MUST** accurately define the length of szErrStr, to prevent a buffer overrun. The underlying API library is implemented in C, which does not provide any means to determine size or rank of the character array.

returns: TCL_OK | TCL_ERROR

NOTE: the only circumstance under which this function should return TCL_ERROR is if szErrStr == NULL.

2.4.10. PowerShell_GetVersion

function: PowerShell_GetVersion (char *pszLibVer, int iLenszLibVer);

description: returns the version string for the library. Will return an error if the buffer passed in is too small (indicated by the iLenszLibVer parameter). Note that at the time this comment was written, the version string was 14 chars long, with the format "1.0 06/24/2009".

parameters: char *pszLibVer - pointer to a location to store the version string in.

int iLenszLibVer - the length of the buffer pointed to by pszLibVer.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.11. PowerShell_ProcessCommand

function: PowerShell_ProcessCommand (char *szCmd, char *szResponse, int iLenResponse);

description: evaluates the command string passed in pszCmd with the Tcl interpreter, and returns the Tcl result in the buffer pszResponse. The string to be evaluated should be a valid Tcl command, and must be able to run to completion - i.e. it cannot be a command that would require any interactive input (for example, the Tcl command “gets stdin”).

parameters: char *pszCmd - pointer to the char buffer containing the command string. This string will be processed with the Tcl “eval” command. Any valid form of Tcl command string should be acceptable, including the use of ‘{‘ and ‘}’ or ‘””’ to delimit list arguments, and ‘[‘ ‘]’ brackets to delimit and embedded command response. Examples:

```
psa_enable ?
set x "1 2 3"
set x {1 2 3}
set y [llength $x]
```

char *pszResponse – pointer to the char buffer pointed to copy the Tcl result into.

NOTE: the Tcl “eval” command returns a Tcl list result. To assist the calling context when parsing this list, each list element is delimited with a ‘;’ character. For example, the response to “psa_enable ?” takes the form:

```
PSA_Chassis=;192.168.221.140;Serial_No=;04AAD7BA5;PSA2400_Ctlr=;DISABLED;
PSA_Interactive=;ENABLED;PSE_Conf_Suite=;ENABLED;Adv_Resources=;ENABLED;
PSE_Multiport_Suite=;ENABLED;LLDP=;ENABLED;PHY_Performance_Suite=;ENABLED
```

int iLenResponse - the length of the buffer pointed to by pszResponse.

NOTE: this parameter **MUST** accurately define the length of szResponse, to prevent a buffer overrun. The underlying API library is implemented in C, which does not provide any means to determine size or rank of the character array. If the Tcl result is longer than iLenResponse, the string being returned will be truncated.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.12. PowerShell_SetDebug

function: PowerShell_SetDebug (char *szDbgFile, int iState);

description: sets an internal variable that controls debug behavior. If the iState parameter == 1, attempts to write a message to the file defined by szDbgFile. Note that the file open mode used in this case is "w", so this is a destructive operation with respect to a pre-existing file of the same name.

The “debug” mode is primarily intended to be used to collect internal library function information to aid Sifos in providing technical support. The information consists of API argument to PowerShell Tcl translation, and result parsing.

parameters: char *szDbgFile - path to the debug file to use.

int iState - 1 = enable debug 0 = disable debug

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.4.13. PowerShell_GetDebug

function: PowerShell_GetDebug(int *piState)

description: returns the current state of the internal debug flag.

parameters: int *piState – location to store the flag state in.

returns: TCL_OK

2.5. PVA_SetConnect

function: PVA_SetConnect (int iSlot, int iPort, ePVAConnect eSetting)

description: configures the RJ-45 connection setting using the PowerShell command "pva_connect".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVAConnect eSetting - PVA_PHY | PVA_THROUGH

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.6. PVA_GetConnect

function: PVA_GetConnect (int iSlot, int iPort, ePVAConnect *peSetting)

description: queries the RJ-45 connection setting using the PowerShell command "pva_connect ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAConnect *peSetting - the location to store the CONNECTION state in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.7. PVA_SetLine

function: PVA_SetLine (int iSlot, int iPort, ePVAImpair ePair12, ePVAImpair ePair34)

description: configures the 100m cat5e passive line simulator setting using the PowerShell command "pva_line".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVAImpair ePair12 - PVA_NORMAL | PVA_IMPAIR

ePVAImpair ePair34 - PVA_NORMAL | PVA_IMPAIR

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.8. PVA_GetLine

function: PVA_GetLine (int iSlot, int iPort, ePVAImpair *pePair12, ePVAImpair *pePair34)

description: queries the 100m cat5e passive line simulator setting using the PowerShell command "pva_line ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAImpair *pePair12 - the location to store the pair12 state in.

ePVAImpair *pePair34 - the location to store the pair34 state in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.9. PVA_SetMismatch

function: PVA_SetMismatch (int iSlot, int iPort, ePVAImpair ePair12, ePVAImpair ePair34)

description: configures the 11.5 dB line mismatch setting using the PowerShell command "pva_mismatch".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVAImpair ePair12 - PVA_NORMAL | PVA_IMPAIR

ePVAImpair ePair34 - PVA_NORMAL | PVA_IMPAIR

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.10. PVA_GetMismatch

function: PVA_GetMismatch (int iSlot, int iPort, ePVAImpair *pePair12, ePVAImpair *pePair34)

description: queries the 11.5 dB line mismatch setting using the PowerShell command "pva_mismatch ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAImpair *pePair12 - the location to store the pair12 state in.

ePVAImpair *pePair34 - the location to store the pair34 state in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.11. PVA_SetNoise

function: PVA_SetNoise (int iSlot, int iPort, eNoiseState eState, double dNoisePower)

description: configures the noise source using the PowerShell command "pva_noise".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

eNoiseState eState - PVA_NOISE_OFF | PVA_NOISE_ON

double dNoisePower - noise level, -6 to +21.5, in 0.5 dB steps.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.12. PVA_GetNoise

function: PVA_GetNoise (int iSlot, int iPort, eNoiseState *peState, double *pdNoisePower)

description: queries the noise source setting using the PowerShell command "pva_noise ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

eNoiseState *peState - location to store the state in

double *pdNoisePower - location to store the noise level setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.13. PVA_SetTxClk

function: PVA_SetTxClk (int iSlot, int iPort, ePVATxClkState eState, int iOffset, double dJitter)

description: configures the Tx clock settings using the PowerShell command "pva_tx_clk".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATxClkState eState - PVA_CLK_NORMAL | PVA_CLK_OFFSET | PVA_CLK_JITTER

int iOffset - -115, -100, -50, 0, 50, 100, 115 ppm.

double dJitter - -6.0 to + 24.0 dB, in 0.5 dB steps.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.14. PVA_GetTxClk

function: PVA_GetTxClk (int iSlot, int iPort, ePVATxClkState *peState, int *piOffset, double *pdJitter)

description: queries the Tx clock settings using the PowerShell command "pva_tx_clk ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATxClkState *peOffsetState - location to store the state in.

int *piOffset - location to store the offset setting in.

ePVATxClkState *peJitterState - location to store the state in.

double *pdJitter - location to store the jitter setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.15. PVA_Reset

function: PVA_Reset (int iSlot, int iPort, ePVABoolean bRestore)

description: commands the test port to perform a reset, by executing the PowerShell command "pva_reset".

parameters: int iSlot - the slot that the blade to reset is installed in. Legal value: 1..12

int iPort - the port on the specified blade to reset. Legal value: 1|2

ePVABoolean bRestore - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.16. PVA_Relink

function: PVA_Relink (int iSlot, int iPort)

description: commands the test port PHY to perform a relink, using the PowerShell command "pva_relink".

parameters: int iSlot - the slot that the blade to relink is installed in. Legal value: 1..12

int iPort - the port on the specified blade to relink. Legal value: 1|2

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.17. PVA_SetSpeed

function: PVA_SetSpeed (int iSlot, int iPort, ePVASpeed eSpeed, ePVADuplex eDuplex, ePVABoolean bArm)

description: configures the test PHY advertised link settings for speed and duplex using the PowerShell command "pva_speed".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVASpeed eSpeed - PVA_10 | PVA_100 | PVA_1000 | PVA_10_100 | PVA_SPEED_AUTO

ePVADuplex eDuplex - PVA_AUTO_DUPLEX | PVA_FULL_DUPLEX | PVA_HALF_DUPLEX

ePVABoolean bArm - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.18. PVA_GetSpeed

function: PVA_GetSpeed (int iSlot, int iPort, ePVASpeed *peSpeed, ePVADuplex *peDuplex)

description: queries the test PHY advertised link settings using the PowerShell command "pva_speed ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVASpeed *peSpeed - the location to put the speed setting in.

ePVADuplex *peDuplex - the location to put the duplex setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.19. PVA_SetPolarity

function: PVA_SetPolarity (int iSlot, int iPort, ePVAPolarity ePolarity, ePVABoolean bArm)

description: configures the test PHY to advertise the specified polarity using the PowerShell command "pva_polarity".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVAPolarity ePolarity - PVA_POL_AUTO | PVA_MDI | PVA_MDIX

ePVABoolean bArm - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.20. PVA_GetPolarity

function: PVA_GetPolarity (int iSlot, int iPort, ePVAPolarity *pePolarity)

description: queries the test PHY advertised polarity setting using the PowerShell command "pva_polarity ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAPolarity *pePolarity - the location to put the polarity setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.21. PVA_SetGigMode

function: PVA_SetGigMode (int iSlot, int iPort, ePVAGigMode eMode, ePVABoolean bArm)

description: configures the test PHY to advertise the specified 1000Base-T Master/Slave mode using the PowerShell command "pva_gig_mode".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVAGigMode eMode - PVA_GIG_AUTO | PVA_GIG_MASTER | PVA_GIG_SLAVE

ePVABoolean bArm - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.22. PVA_GetGigMode

function: PVA_GetGigMode (int iSlot, int iPort, ePVAGigMode *peMode)

description: queries the test PHY advertised gig mode setting using the PowerShell command "pva_gig_mode ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAGigMode *peMode - the location to put the gig mode setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.23. PVA_GetLinkState

function: PVA_GetLinkState (int iSlot, int iPort, ePVALinkState *peState)

description: queries the test PHY for the current link state using the PowerShell command "pva_get_link_state".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVALinkState *peState - the location to put the link state in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.24. PVA_GetLinkSpeed

function: PVA_GetLinkSpeed (int iSlot, int iPort, ePVASpeed *peSpeed)

description: queries the test PHY for the current link speed using the PowerShell command "pva_get_link_speed".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVASpeed *peSpeed - the location to put the link speed in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.25. PVA_GetGigState

function: PVA_GetGigState (int iSlot, int iPort, ePVAGigMode *peMode)

description: queries the test PHY for the current gig mode using the PowerShell command "pva_get_gig_state".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVAGigMode *peMode - the location to put the gig mode in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.26. PVA_SetRxGain

function: PVA_SetRxGain (int iSlot, int iPort, int iLevel)

description: configures the test PHY RxGain setting using the PowerShell command "pva_rx_gain".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

int iLevel - 1..10. Use 0 for AUTO.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.27. PVA_GetRxGain

function: PVA_GetRxGain (int iSlot, int iPort, int *piLevel)

description: queries the test PHY RxGain setting using the PowerShell command "pva_rx_gain ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

int *piLevel - the location to store the gain setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.28. PVA_SetTxGain

function: PVA_SetTxGain (int iSlot, int iPort, int iLevel)

description: configures the test PHY TxGain setting using the PowerShell command "pva_tx_gain".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

int iLevel - 1..10. Use 0 for AUTO.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.29. PVA_GetTxGain

function: PVA_GetTxGain (int iSlot, int iPort, int *piLevel)

description: queries the test PHY TxGain setting using the PowerShell command "pva_tx_gain ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

int *piLevel - the location to store the gain setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.30. PVA_SetTxSlew

function: PVA_SetTxSlew (int iSlot, int iPort, int iRate)

description: configures the test PHY Tx Slew setting using the PowerShell command "pva_tx_slew".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

int iRate - 1..10. Use 0 for AUTO.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.31. PVA_GetTxSlew

function: PVA_GetTxSlew (int iSlot, int iPort, int *piRate)

description: queries the test PHY Tx Slew setting using the PowerShell command "pva_tx_slew ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

int *piRate - the location to store the slew rate setting in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.32. PVA_DoCals

function: PVA_DoCals (char *pszPortList, int iNumPorts, int iMeasTypes, int iLinkSpeeds, int iNumAvgs)

description: initiates calibrations using the PowerShell command "pva_cals".

parameters: char *pszPortList - a string containing a list of valid PowerShell PSA test blade port specifications, separated by spaces. For example: "1,1 1,2 5,2 7,1"
As defined, only a single port for any given slot can be calibrated at one time, since the adjacent port on that test blade is used as the cal partner.

int iNumPorts - the number of unique ports in the szPortList string.

int iMeasTypes - valid ePVAMeasTypes enums. This may be a single measurement type enum, or any combination of measurement type enums, or'ed together.

PVA_MEAS_PSD | PVA_MEAS_ECHO | PVA_MEAS_XTALK

int iLinkSpeeds - valid ePVASpeed enums. This may be a single link speed enum, or a combination of link speed enums, or'ed together.

PVA_100 | PVA_1000

int iNumAvgs - number of averages to use when making the calibration measurements.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.33. PVA_ShowCals

function: PVA_ShowCals (char *pszChassisAddr, char *pszBuf, int iLenBuf)

description: returns the list of cal files available for the defined chassis produced by the PowerShell command "pva_show_cal".

parameters: char *pszChassisAddr - IP address of the chassis to check for available calibration files.

char *pszBuf - string to copy the result into.

int iLenBuf – the length (in characters) of szBuf

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.34. PVA_MeasRxPower

function: PVA_MeasRxPower (int iSlot, int iPort, ePVAPair *pePair, ePVALinkState *peState, double *pdMeas)

description: performs an Rx power measurement using the PowerShell command "pva_rx_pwr stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAPair *pePair - for input purposes, this location should be set to a legal pair enumeration:

PVA_PAIR1 | PVA_PAIR2 | PVA_PAIR3 | PVA_PAIR4

When the function returns, the variable will be set to the pair enumeration for the actual pair measured. For 1000BaseT, this will always be the same as the pair defined by the caller. However, for 100BaseT, the meter will automatically report the power for the actual pair being used as the Tx pair by the DUT, which may not be the pair that the caller specified.

ePVALinkState *peState - the location to put the link state in.

double *pdMeas - the location to store the measured Rx power in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.35. PVA_MeasSkew

function: PVA_MeasSkew (int iSlot, int iPort, ePVAMeterStatus *peStatus, int *piPair1Skew, int *piPair2Skew, int *piPair3Skew, int *piPair4Skew)

description: performs a timing skew measurement using the PowerShellcommand "pva_skew stat".

parameters: int iSlot - the slot that the blade to measure with isinstalled in. Legal value: 1..12

int iPort - the port on the specified blade to measurewith. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to put themeter status in.

int *piPair1Skew - the location to store the measuredpair 1 skew in.

int *piPair2Skew - the location to store the measuredpair 2 skew in.

int *piPair3Skew - the location to store the measuredpair 3 skew in.

int *piPair4Skew - the location to store the measuredpair 4 skew in.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.36. PVA_SetConfigSNR

function: PVA_SetConfigSNR (int iSlot, int iPort, ePVATrig eTrig, ePVATimeout eTimeout, ePVASpeed eSpeed, ePVAPair ePair, int iNumAvg)

description: configures the SNR measurement settings using the PowerShell command "pva_snr".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATrig eTrig - PVA_TRIG_OFF | PVA_TRIG_EXTERNAL

ePVATimeout eTimeout - PVA_TIMEOUT_10S | PVA_TIMEOUT_100S

ePVASpeed eSpeed - PVA_100 | PVA_1000

ePVAPair ePair - PVA_PAIR1 | PVA_PAIR2 | PVA_PAIR3 | PVA_PAIR4

int iNumAvg - 1..64

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.37. PVA_GetConfigSNR

function: PVA_GetConfigSNR (int iSlot, int iPort, ePVATrig *peTrig, ePVATimeout *peTimeout, ePVASpeed *peSpeed, ePVAPair *pePair, int *piNumAvg)

description: queries the SNR measurement settings using the PowerShell command "pva_snr ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATrig *peTrig - location to store the trigger setting

ePVATimeout *peTimeout - location to store the timeout setting

ePVASpeed *peSpeed - location to store the speed setting

ePVAPair *pePair - location to store the pair setting

int *piNumAvg - location to store the avg setting

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.38. PVA_GetStatusSNR

function: PVA_GetStatusSNR (int iSlot, int iPort, ePVAMeterStatus *peStatus, ePVAPair *pePair, double *pdMeas)

description: performs an SNR measurement using the PowerShell command "pva_snr stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to store the meter status.

ePVAPair *pePair - the location to store the measured pair identifier.

double *pdMeas - the location to store the measured SNR.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.39. PVA_SetConfigPSD

function: PVA_SetConfigPSD (int iSlot, int iPort, ePVATrig eTrig, ePVATimeout eTimeout, ePVASpeed eSpeed, ePVAPair ePair, int iNumAvg, double dStartFreq, double dStopFreq)

description: configures the PSD measurement settings using the PowerShell command "pva_psd".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATrig eTrig - PVA_TRIG_OFF | PVA_TRIG_EXTERNAL

ePVATimeout eTimeout - PVA_TIMEOUT_10S | PVA_TIMEOUT_100S

ePVASpeed eSpeed - PVA_100 | PVA_1000

ePVAPair ePair - PVA_PAIR1 | PVA_PAIR2 | PVA_PAIR3 | PVA_PAIR4

int iNumAvg - 1..64

double dStartFreq - 0.02 MHz - 80.0 MHz

double dStopFreq - 0.2 MHz - 100.0 MHz

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.40. PVA_GetConfigPSD

function: PVA_GetConfigPSD (int iSlot, int iPort, ePVATrig *peTrig, ePVATimeout *peTimeout,
ePVASpeed *peSpeed, ePVAPair *pePair, int *piNumAvg, double *pdStartFreq,
double *pdStopFreq)

description: queries the PSD measurement settings using the PowerShell command "pva_psd ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATrig *peTrig - location to store the trigger setting

ePVATimeout *peTimeout - location to store the timeout setting

ePVASpeed *peSpeed - location to store the speed setting

ePVAPair *pePair - location to store the pair setting

int *piNumAvg - location to store the avg setting

double *pdStartFreq - location to store the start setting

double *pdStopFreq - location to store the stop setting

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.41. PVA_GetStatusPSD

function: PVA_GetStatusPSD (int iSlot, int iPort, ePVAMeterStatus *peStatus, ePVAPair *pePair, double *pdFreq, double *pdMeas)

description: performs a PSD measurement using the PowerShell command "pva_psd stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to store the meter status.

ePVAPair *pePair - the location to store the measured pair identifier.

double *pdFreq - the location to store the measured PSD trace frequency points.

NOTE: the caller must provide a contiguous array that contains at least 33 sizeof(double) elements.

double *pdMeas - the location to store the measured PSD trace data.

NOTE: the caller must provide a contiguous array that contains at least 33 sizeof(double) elements.

NOTE: if the arrays are not sufficiently large enough a run time memory access error will very likely occur.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.42. PVA_SetConfigECHO

function: PVA_SetConfigECHO (int iSlot, int iPort, ePVATrig eTrig, ePVATimeout eTimeout, ePVAPair ePair, int iNumAvg)

description: configures the ECHO measurement settings using the PowerShell command "pva_echo".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATrig eTrig - PVA_TRIG_OFF | PVA_TRIG_EXTERNAL

ePVATimeout eTimeout - PVA_TIMEOUT_10S | PVA_TIMEOUT_100S

ePVAPair ePair - PVA_PAIR1 | PVA_PAIR2 | PVA_PAIR3 | PVA_PAIR4

int iNumAvg - 1..64

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.43. PVA_GetConfigECHO

function: PVA_GetConfigECHO (int iSlot, int iPort, ePVATrig *peTrig, ePVATimeout *peTimeout, ePVAPair *pePair, int *piNumAvg)

description: queries the ECHO measurement settings using the PowerShell command "pva_echo ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATrig *peTrig - location to store the trigger setting

ePVATimeout *peTimeout - location to store the timeout setting

ePVAPair *pePair - location to store the pair setting

int *piNumAvg - location to store the avg setting

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.44. PVA_GetStatusECHO

function: PVA_GetStatusECHO (int iSlot, int iPort, ePVAMeterStatus *peStatus, ePVAPair *pePair, double *pdMeas)

description: performs an ECHO measurement using the PowerShell command "pva_echo stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to store the meter status.

ePVAPair *pePair - the location to store the measured pair identifier.

double *pdMeas - the location to store the measured ECHO.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.45. PVA_SetConfigXTALK

function: PVA_SetConfigXTALK (int iSlot, int iPort, ePVATrig eTrig, ePVATimeout eTimeout, ePVAPair ePair, int iNumAvg)

description: configures the XTALK measurement settings using the PowerShell command "pva_xtalk".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATrig eTrig - PVA_TRIG_OFF | PVA_TRIG_EXTERNAL

ePVATimeout eTimeout - PVA_TIMEOUT_10S | PVA_TIMEOUT_100S

ePVAPairGroup ePairGroup - PVA_PAIRS12 | PVA_PAIRS13 | PVA_PAIRS14 | PVA_PAIRS23 | PVA_PAIRS24 | PVA_PAIRS34

int iNumAvg - 1..64

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.46. PVA_GetConfigXTALK

function: PVA_GetConfigXTALK GetConfigECHO (int iSlot, int iPort, ePVATrig *peTrig, ePVATimeout *peTimeout, ePVAPair *pePair, int *piNumAvg)

description: queries the XTALK measurement settings using the PowerShell command "pva_xtalk ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATrig *peTrig - location to store the trigger setting

ePVATimeout *peTimeout - location to store the timeout setting.

ePVAPairGroup *pePairGroup - location to store the pair group setting.

int *piNumAvg - location to store the avg setting

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.47. PVA_GetStatusXTALK

function: PVA_GetStatusXTALK (int iSlot, int iPort, ePVAMeterStatus *peStatus, ePVAPair *pePair, double *pdMeas)

description: performs an XTALK measurement using the PowerShell command "pva_xtalk stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to store the meter status.

ePVAPair *pePair - the location to store the measured pair identifier.

double *pdMeas - the location to store the measured XTALK.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.48. PVA_SetConfigLinkMon

function: PVA_SetConfigLinkMon (int iSlot, int iPort, ePVATrig eTrig, ePVATimeout eTimeout, ePVASample eSample, ePVAPeriod ePeriod, int iCount, ePVABoolean bClear)

description: configures the Link Monitor measurement settings using the PowerShell command "pva_link_mon".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVATrig eTrig - PVA_TRIG_OFF | PVA_TRIG_EXTERNAL

ePVATimeout eTimeout - PVA_TIMEOUT_10S | PVA_TIMEOUT_100S

ePVASample eSample - PVA_LINK | PVA_REMOTE | PVA_LOCAL

ePVAPeriod ePeriod - PVA_20MSEC | PVA_50MSEC | PVA_100MSEC

int iCount - 1..100

ePVABoolean bClear - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.49. PVA_GetConfigLinkMon

function: PVA_GetConfigLinkMon (int iSlot, int iPort, ePVATrig *peTrig, ePVATimeout *peTimeout, ePVASample *peSample, ePVAPeriod *pePeriod, int *piCount)

description: queries the Link Monitor measurement settings using the PowerShell command "pva_link_mon ?".

parameters: int iSlot - the slot that the blade to query is installed in. Legal value: 1..12

int iPort - the port on the specified blade to query. Legal value: 1|2

ePVATrig *peTrig - location to store the trigger setting

ePVATimeout *peTimeout - location to store the timeout setting.

ePVASample *peSample - location to store the sample setting.

ePVAPeriod *pePeriod - location to store the period setting.

int *piCount - location to store the count setting.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.50. PVA_GetStatusLinkMon

function: PVA_GetStatusLinkMon (int iSlot, int iPort, ePVAMeterStatus *peStatus, ePVALinkState *peLinkState, ePVASpeed *pdSpeed, ePVAPolarity *pePolarity, ePVADuplex *peDuplex, ePVAGigMode *peMode, int *piCfgCount, ePVASample *peSample, int *piMeasCount, ePVAPair14State *peState)

description: performs a Link Monitor measurement using the PowerShell command "pva_link_mon stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVAMeterStatus *peStatus - the location to store the meter status.

ePVALinkState *peLinkState - the location to store the measured link state.

ePVASpeed *pdSpeed - the location to store the measured link speed.

ePVAPolarity *pePolarity - the location to store the measured polarity.

ePVADuplex *peDuplex - the location to store the measured duplex.

ePVAGigMode *peMode - the location to store the measured gig mode.

int *piCfgCount - the location to store the configured count.

ePVASample *peSample - location to store the sample setting.

int *piMeasCount - the location to store the measured count.

ePVAPair14State *peState - the location to store the measured pair 1-4 state.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.51. PVA_SetConfigTxPkt

function: PVA_SetConfigTxPkt (int iSlot, int iPort, int iFrameSize, ePVAFrameCount eCount, ePVATxRate eRate, char *pszPayload, ePVABoolean bStart)

description: configures the Packet Transmitter settings using the PowerShell command "pva_tx_pkt".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

int iFrameSize - 16..1512

ePVAFrameCount eCount - PVA_32K | PVA_128K | PVA_512K | PVA_1024K | PVA_CONTINUOUS

ePVATxRate eRate - PVA_LINE | PVA_MED | PVA_SLOW

char *pszPayload - exactly 8 hex characters

ePVABoolean bStart - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.52. PVA_GetConfigTxPkt

function: PVA_GetConfigTxPkt (int iSlot, int iPort, int *piFrameSize, ePVAFrameCount *peCount, ePVATxRate *peRate, char *pszPayload)

description: queries the Packet Transmitter settings usingthe PowerShell command "pva_tx_pkt ?".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

int *piFrameSize - the location to store the frame size

ePVAFrameCount *peCount - the location to store the frame count.

ePVATxRate *peRate - the location to store the frame rate

char *pszPayload - the location to store the payload

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.53. PVA_GetStatusTxPkt

function: PVA_GetStatusTxPkt (int iSlot, int iPort, ePVATxStatus *peStatus)

description: returns the status of the Packet Transmitter using the PowerShell command "pva_tx_pkt stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVATxStatus *peStatus - the location to store the transmitter status.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.54. PVA_SetConfigRxPkt

function: PVA_SetConfigRxPkt (int iSlot, int iPort, ePVABoolean bStart)

description: controls the Packet Receiver using the PowerShell command "pva_rx_pkt".

parameters: int iSlot - the slot that the blade to set is installed in. Legal value: 1..12

int iPort - the port on the specified blade to set. Legal value: 1|2

ePVABoolean bStart - PVA_TRUE | PVA_FALSE

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.55. PVA_GetStatusRxPkt

function: PVA_GetStatusRxPkt (int iSlot, int iPort, ePVARxStatus *peStatus, unsigned long *plCount, unsigned long *plCRCerr)

description: returns the status of the Packet Receiver using the PowerShell command "pva_rx_pkt stat".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVARxStatus *peStatus - the location to store the receiver status.

unsigned long *plCount - the location to store the rx count.

unsigned long *plCRCerr - the location to store the CRC error count.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.56. PVA_GetStatusPartner

function: PVA_GetStatusPartner (int iSlot, int iPort, ePVALinkState *peLinkState, ePVAAutoNeg *peAuto, ePVAAutoNeg *peAck, ePVADuplexSupport *peDuplex1000, ePVADuplexSupport *peDuplex100, ePVADuplexSupport *peDuplex10, ePVABoolean *pe100T4, ePVAPause *pePause, ePVALinkOK *peLinkOK, ePVARxOK *peRxOK, ePVAPolarity *pePolarity, ePVAGigMode *peMode)

description: performs a PVA Partner measurement using the PowerShell command "pva_partner stat". This function has been superseded by PVA_GetStatusPartner2, which provides access to the new "NLP_LINK" parameter that was added to the result in PowerShell PSA version 4.1.0. This older version was retained to provide compatibility for any pre-existing application.

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVALinkState *peLinkState - the location to store the measured link state.

ePVAAutoNeg *peAuto - the location to store the auto negotiation response.

ePVAAutoNeg *peAck - the location to store the auto negotiation ack response.

ePVADuplexSupport *peDuplex1000 - the location to store the 1000BaseT capability.

ePVADuplexSupport *peDuplex100 - the location to store the 100BaseT capability.

ePVADuplexSupport *peDuplex10 - the location to store the 10BaseT capability.

ePVABoolean *pe100T4 - the location to store whether or not the partner is 100BaseT4 capable.

ePVAPause *pePause - the location to store the pause capability.

ePVALinkOK *peLinkOK - the location to store the measured link status.

ePVARxOK *peRxOK - the location to store the measured rx status.

ePVAPolarity *pePolarity - the location to store the measured polarity.

ePVAGigMode *peMode - the location to store the advertised gig mode.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.57. PVA_GetStatusPartner2

function: PVA_GetStatusPartner2 (int iSlot, int iPort, ePVALinkState *peLinkState, ePVAAutoNeg *peAuto, ePVAAutoNeg *peAck, ePVADuplexSupport *peDuplex1000, ePVADuplexSupport *peDuplex100, ePVADuplexSupport *peDuplex10, ePVABoolean *pe100T4, ePVAPause *pePause, ePVALinkOK *peLinkOK, ePVARxOK *peRxOK, ePVAPolarity *pePolarity, ePVAGigMode *peMode, ePVALinkState * peNLPLinkState)

description: performs a PVA Partner measurement using the PowerShell command "pva_partner stat". This function provides access to the new "NLP_LINK" parameter that was added to the result in PowerShell PSA version 4.1.0.

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

ePVALinkState *peLinkState - the location to store the measured link state.

ePVAAutoNeg *peAuto - the location to store the auto negotiation response.

ePVAAutoNeg *peAck - the location to store the auto negotiation ack response.

ePVADuplexSupport *peDuplex1000 - the location to store the 1000BaseT capability.

ePVADuplexSupport *peDuplex100 - the location to store the 100BaseT capability.

ePVADuplexSupport *peDuplex10 - the location to store the 10BaseT capability.

ePVABoolean *pe100T4 - the location to store whether or not the partner is 100BaseT4 capable.

ePVAPause *pePause - the location to store the pause capability.

ePVALinkOK *peLinkOK - the location to store the measured link status.

ePVARxOK *peRxOK - the location to store the measured rx status.

ePVAPolarity *pePolarity - the location to store the measured polarity.

ePVAGigMode *peMode - the location to store the advertised gig mode.

ePVALinkState * peNLPLinkState - the location to store the measured NLP link state.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.58. PVA_ConnCheck

function: PVA_ConnCheck (char *pszPortList, int iNumPorts, ePVABoolean *peStatus)

description: performs a connection check using the PowerShell command "pva_conn_check".

parameters: char *pszPortList - a string containing a list of valid PowerShell PSA test blade port specifications, separated by spaces. For example:

"1,1 1,2 5,2 7,1"

NOTE: each test port needs to be connected to a link partner.

int iNumPorts - the number of unique ports in the szPortList string.

ePVABoolean *peStatus - the location to store the result per port.

NOTE: eStatus needs to be an array that is (iNumPorts * sizeof(int)) long. The array needs to be organized as a contiguous block of memory.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.59. PVA_LinkWait

function: PVA_LinkWait (int iSlot, int iPort, int iTimeout, ePVALinkState *peLinkState)

description: monitors for a link to come up using the PowerShell command "pva_link_wait".

parameters: int iSlot - the slot that the blade to measure with is installed in. Legal value: 1..12

int iPort - the port on the specified blade to measure with. Legal value: 1|2

int iTimeout - 5..30

ePVALinkState *peLinkState - the location to store the link state.

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

2.60. PVA_Sequence

function: PVA_Sequence ConnCheck (char *pszPortList, char *pszTestList, char *pszName, ePVABoolean bPackets, ePVABoolean bBreak, ePVABoolean bInhibit, ePVAReportType eReportType)

description: runs a PHY Tests sequence using the PowerShell command "pva_sequence".

parameters: char *pszPortList - a string containing a list of valid PowerShell PSA test blade port specifications, separated by spaces. For example:

"1,1 1,2 5,2 7,1"

Use the test name "all" to have the sequencer use all available PVA test ports.

char *pszTestList - a string containing a list of valid PHY Test Suite test names. Use the test name "all" to access the default behavior of running all tests.

char *pszName - a string containing used defined text to be used on the PHY Test Suite produced report.

ePVABoolean bPackets - PVA_TRUE | PVA_FALSE

ePVABoolean bBreak - PVA_TRUE | PVA_FALSE

ePVABoolean bInhibit - PVA_TRUE | PVA_FALSE

ePVAReportType eReportType - PVA_SPREADSHEET_REPORT | PVA_TEXT_REPORT

returns: TCL_OK | TCL_ERROR

If an error occurs, an associated error message can be retrieved by calling PowerShell_GetErrString.

3. PhyView API Definitions for Visual Basic and C#

3.1. Visual Basic 6 Type Differences

There are type differences in VB6 that you will need to keep in mind:

API Type	VB6 Type
int	Long
char *	String

The VB6 boolean does not pass correctly, so for any API call that requires a boolean, you must pass an integer:

0 = false 1 = true

Special consideration must be given to passing strings that need to be written to by the API function. The string MUST be declared using the form of the dim statement that defines the length. For example, the following declaration creates a string that is 512 characters long:

dim szErrStr as string * 512

The DLL runtime has no way to determine the length of space that has actually been allocated for a string (unlike the VARIANT type mechanism that is native to VB6), so for any API call that includes a string that will be written to by the API function, there will be a corresponding argument that is used to pass the length of the string to the API function.

The value passed to the API function MUST accurately define the length, or you WILL encounter runtime errors related to illegal memory location access.

3.2. Visual Basic.NET Type Differences

There are type differences in VB .NET that you will need to keep in mind:

API Type	VB .NET Type
int	Integer
char *	System.Text.StringBuilder

Special consideration must be given to passing strings that need to be written to by the API function. The string MUST be declared using the System.Text.StringBuilder object, with a defined length. For example:

```
Dim strBuffer As System.Text.StringBuilder = _
   New System.Text.StringBuilder(LEN_PWRSHELL_ERR_STR_BUF)
```

The DLL runtime has no way to determine the length of space that has actually been allocated for a string, so for any API call that includes a string that will be written to by the API function, there will be a corresponding argument that is used to pass the length of the string to the API function.

The value passed to the API function MUST accurately define the length, or you WILL encounter runtime errors related to illegal memory location access.

3.3. Enumerations

```
Public Const TCL_OK As Long = 0
Public Const TCL_ERROR As Long = 1

Public Enum ePSABoolean
   PSA_FALSE = 0
   PSA_TRUE = 1
End Enum

Public Enum eInitStatus
   PSA_INITIALIZED = 1
   PSA_NOT_INITIALIZED = 2
End Enum

Public Enum eAltSetting
   ALT_UNKNOWN = 0
   ALT_A = 1
   ALT_B = 2
End Enum

Public Enum ePolSetting
   POL_UNKNOWN = 0
   POL_POS = 1
   POL_NEG = 2
End Enum

Public Enum eTestRsltsDest
   TEST_RESULTS_DEST_UNKNOWN = 0
   TEST_RESULTS_DEST_SPREADSHEET = 1
   TEST_RESULTS_DEST_TEXT = 2
End Enum
```

3.4. Visual Basic 6 Service Functions

The service function TrimAPIString provides the means to trim any unused space off of the end of a fixed length string containing a value passed back by a PhyViewAPI function. This function is defined in the source module PowerShellAPI.bas.

3.5. Visual Basic 6 Compatible Declarations

Visual Basic 6 compatible declarations are located in the source module PhyViewAPI.bas. The declarations for functions that are shared with the PowerSync Analyzer are located in the source module PowerShellAPI.bas.

Parameters that are only passed in to API functions, and are not used to return values, are declared as being passed “ByVal”.

Most parameters that are passed to API functions for the purpose of being used to return values are declared as being passed “ByRef”. The exception to this is when passing strings. As discussed above in Type Differences, it is necessary to use fixed length strings allocated in VB as the storage location for any value that is to be returned by the API library. These strings are passed “ByVal”, even though they are being written to by the underlying library functions.

For example, this function accepts strings as inputs, but does not write to those strings:

```
Declare Function PowerShell_Init Lib "PowerShellAPI.dll" (ByVal szChassisIPAddress As String, _
        ByVal szRCFile As String) As Long
```

This function writes characters to the string that is passed in:

```
Declare Function PowerShell_GetErrString Lib "PowerShellAPI.dll" (ByVal szErrStr As String, _
        ByVal lLenErrStr As Long) As Long
```

This function reads the value of the enumeration variable passed by value:

```
Declare Function PowerShell_SetAlt Lib "PowerShellAPI.dll" (ByVal lSlot As Long, _
        ByVal lPort As Long, _
        ByVal eSetting As eAltSetting) As Long
```

This function writes a value to the enumeration variable passed by reference:

```
Declare Function PowerShell_GetAlt Lib "PowerShellAPI.dll" (ByVal lSlot As Long, _
        ByVal lPort As Long, _
        ByRef eSetting As eAltSetting) As Long
```

3.6. Visual Basic.NET Compatible Declarations

Visual Basic .NET compatible declarations are located in the source module PhyViewAPI.vb. The declarations for functions that are shared with the PowerSync Analyzer are located in the source module PowerShellAPI.vb.

Parameters that are only passed in to API functions, and are not used to return values, are declared as being passed “ByVal”.

Most parameters that are passed to API functions for the purpose of being used to return values are declared as being passed “ByRef”. The exception to this is when passing strings. As discussed above in Type Differences, it is necessary to use fixed length strings allocated in VB as the storage location for any value that is to be returned by the API library. These strings are passed “ByVal”, even though they are being written to by the underlying library functions.

For example, this function accepts strings as inputs, but does not write to those strings:

```
Declare Function PowerShell_Init Lib "PowerShellAPI.dll" (ByVal szIPAddress As System.Text.StringBuilder, _
        ByVal szRCFile As System.Text.StringBuilder) As Integer
```

This function writes characters to the string that is passed in:

```
Declare Function PowerShell_GetErrString Lib "PowerShellAPI.dll" (ByVal szErrStr As System.Text.StringBuilder, _
        ByVal iLenErrStr As Integer) As Integer
```

The variable to be passed as “szErrStr” needs to be declared with space already allocated:

```
dim szErrStr as System.Text.StringBuilder = New System.Text.StringBuilder (LEN_PWRSHELL_ERR_STR_BUF)
```

This function reads the value of the enumeration variable “eSetting”, which is passed by value:

```
Declare Function PVA_SetConnect Lib "PowerShellAPI.dll" (ByVal iSlot As Integer, _
                ByVal iPort As Integer, _
                ByVal eSetting As ePVAConnect) As Integer
```

This function writes a value to the enumeration variable “eSetting”, which is passed by reference:

```
Declare Function PVA_GetConnect Lib "PowerShellAPI.dll" (ByVal iSlot As Integer, _
                ByVal iPort As Integer, _
                ByRef eSetting As ePVAConnect) As Integer
```

3.7. C# Type Differences

There are type differences in C# that you will need to keep in mind:

API Type	C# Type
int	Int32
char *	System.Text.StringBuilder

Special consideration must be given to passing strings that need to be written to by the API function. The string MUST be declared using the System.Text.StringBuilder object, with a defined length. For example:

```
Dim strBuffer As System.Text.StringBuilder = _
   New System.Text.StringBuilder(LEN_PWRSHELL_ERR_STR_BUF)
```

The DLL runtime has no way to determine the length of space that has actually been allocated for a string, so for any API call that includes a string that will be written to by the API function, there will be a corresponding argument that is used to pass the length of the string to the API function.

The value passed to the API function MUST accurately define the length, or you WILL encounter runtime errors related to illegal memory location access.

A number of methods that perform configuration or status queries include the “ref” modifier keyword for parameters that will be filled in by the underlying API function. When these methods are called, the calling context must include the “ref” keyword with the related parameters.

3.8. C# API Class

A C# .NET class that defines enumerations and methods for each of the API functions is located in the source module PhyViewAPI.cs.

4. PhyView API Definitions for LabView

The functions in PowerShellAPI.dll are accessed via VIs which contain **Call Library Function** nodes wired up to LabView objects that represent the correct type for each terminal. Individual VIs have been created for each API function.

4.1. Type Differences

The table below shows the cross reference of API to LabView types that you will need to keep in mind when defining pass parameters on the **Parameters** tab of the **Call Library Function** Configure… dialog:

API Type	LabView Type	LabView Data Type	Pass
double	Numeric	8-byte Double	Value (for set function) Pointer to Value (for meas function)
int	Numeric	Signed 32-bit Integer or Unsigned 32-bit Int.	Value (for set function) Pointer to Value (for get function)
char *	String	C String Pointer	

C char array parameters are defined for the **Call Library Function** as **Type** = String, **String Format** = C String Pointer.

When passing a string that needs to be written to by an API function, the **Minimum Size** field **MUST** be used to define the length of that string.

The DLL runtime has no way to determine the length of space that has actually been allocated for a string, so for any API call that includes a string that will be written to by the API function, there will be a corresponding argument that is used to pass the length of the string to the API function.

The value passed to the API function **MUST** accurately define the length, or you will most likely encounter runtime errors related to illegal memory location access. For example, the size to use for the LabView string passed to the function `PowerShell_GetErrString` is defined by the macro `LEN_PWRSHELL_ERR_STR_BUF`, located in the header file `PowerShellAPI.h`.

When the parameter being passed to an API function is an array, it is also necessary to correctly specify the length of that array when the associated parameter is defined on the **Parameters** tab of the **Call Library Function** dialog.

4.2. Enumerations

The LabView **Enum** control can be used to define an enumeration to be passed to an API function. The **Enum Data Range Representation** can be set to **Unsigned Long** (**U32**), and the **Edit Items** tab on the **Enum Properties** dialog can be used to define the enumeration names and associated values. The enumerations used by the API library are defined in the header file `PhyViewAPI.h`.

NOTE: the LabView **Enum** control always includes a definition for the value 0. In the LabView example furnished by Sifos, in any case where 0 is not defined for a given API function parameter, the value 0 in the LabView **Enum** control is identified with the name “UNKNOWN”.

4.3. Calling Convention

The “Function” tab on the **Call Library Function** properties dialog allows the Calling convention to be specified. This must be set to `stdcall (WINAPI)`.

4.4. LabView and C Function Prototype Comparison

C prototype:

```
int CALLCONV PowerShell_Init(char *szChassisIPAddress, char *szRCFile);
```

LabView prototype (as shown in the **Function prototype** field on the "Function" tab of the **Call Library Function** properties dialog:

```
long PowerShell_Init(CStr ChassisIPAddress, CStr RCFile);
```

C prototype:

```
int CALLCONV PVA_SetMismatch (int iSlot, int iPort, ePVAImpair Pair12,
ePVAImpair Pair34);
```

LabView prototype:

```
long PVA_SetMismatch(unsigned long Slot, unsigned long Port, unsigned
long Pair12, unsigned long Pair34);
```

C prototype (the variables that the Pair12 and Pair34 settings are stored in must be passed as pointers, also referred to as pass by reference):

```
int CALLCONV PVA_GetMismatch (int iSlot, int iPort, ePVAImpair *pPair12,
ePVAImpair *pPair34);
```

LabView prototype (in this case, note the use of the '*' indicating parameters that are being passed by reference):

```
long PVA_GetMismatch(unsigned long Slot, unsigned long Port, unsigned
long *Pair12, unsigned long *Pair34);
```

When an API parameter is defined as pass by reference, the resulting **Call Library Function** node in LabView must have a node of the correct type and size connected to the related INPUT terminals of the Call Library Function node. In the case of the `PowerShell_GetPassiveLoad` function shown above, a Simple Numeric node is connected to the terminals of the Call Library Function node that are designated for Pair12 and Pair34.

4.5. LabView DLL Loading/Unloading Behavior

When a LabView program is loaded and executed for the first time, the API DLL is loaded into memory (specifically, memory associated with the LabView.exe process). If LabView is kept running, the DLL stays loaded for the lifetime of that LabView process, and is not unloaded until LabView exits.

The `PowerShell_Init` function is used to initialize the PowerShell PSA software environment, loading Tcl and connecting to a PSA chassis. This function can only be called once during the lifetime of the LabView process using the API DLL. If the `PowerShell_Init` function is called again, it will return an error. The function `PowerShell_GetInitStatus` should be used to test for whether or not the init function needs to be called.

If the LabView program needs to connect to a different PSA chassis, the function `PowerShell_ConnectToChassis` can be used at any time.

There is a known issue when using API library version 4.1.7 or older with LabView, where an error is indicated when the API library is unloaded. This actually occurs when the Tcl DLL that the API library by necessity uses is unloaded. It has been determined that there is an invalid memory access exception occurring in the Tcl library, only at unload time. The API library is implemented in C++, and has the Tcl library unload wrapped in a try/catch block. This correctly handles the error for all other runtime environments that the API library has been used with to-date. The error mechanism in LabView is unexpected, and not understood at this time. **UPDATE:** This behavior was corrected in API library version 4.1.8, which has no known interop issues with LabView.

4.6. LabView VIs and Sequence Control

LabView VIs for each of the API functions are located under the path: \Example_Code\PhyViewAPI\LabView in the API Library .ZIP archive.

NOTE: these VIs were created with LabView 8.20.

Each VI contains output terminals for inputs that are required by all VIs, to facilitate wiring up VIs in a LabView program. These terminals include `DLL Path`, `Slot`, and `Port`. An API VI example Front Panel is shown in Figure 4.6-1, and the associated Block Diagram in Figure 4.6-2.

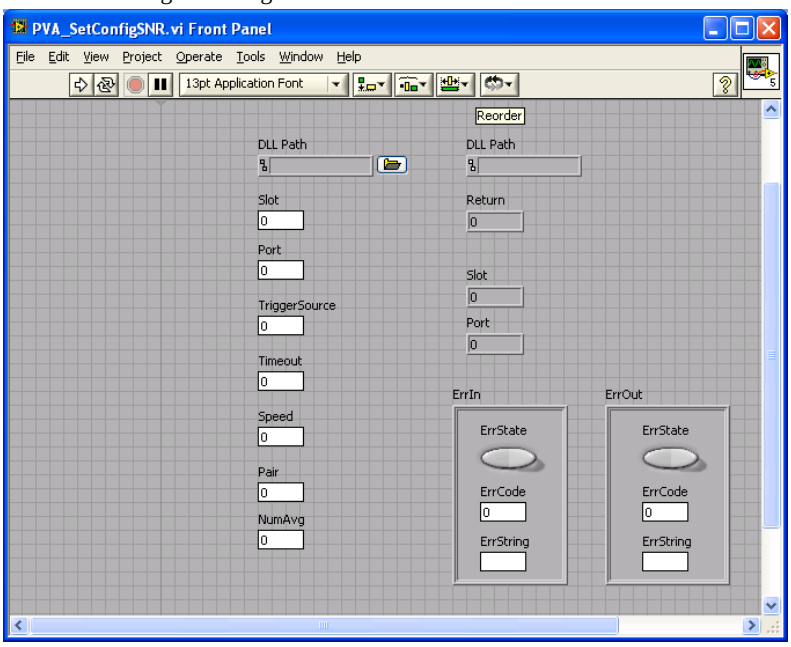

Figure 4.6-1

The various functions in the API library can only be called one at a time, and there are groups of functions that must be called in a specific order. The API VIs do not natively exhibit the data dependent behavior that LabView objects need to exhibit to implicitly control execute sequence. For example, it is necessary to configure a meter before attempting to perform a measurement with that meter.

In order to facilitate the correct sequencing of the API VIs, the `error in` and `error out` terminals of the **Call Library Function** node are brought out to error cluster terminals of each VI. These terminals can be connected between objects to insure that they will sequence in the exact order desired. This is illustrated in Figure 4.6-3.

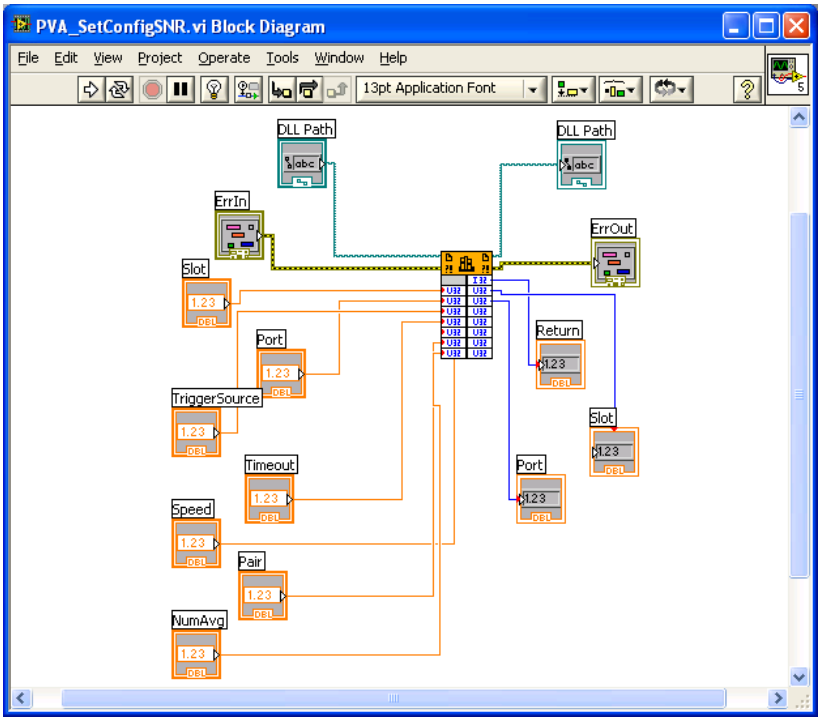

Figure 4.6-2

The example `PhyViewAPI_LabView_Example.vi` shows how to use various API functions, and how to wire them up to achieve a desired sequence flow.

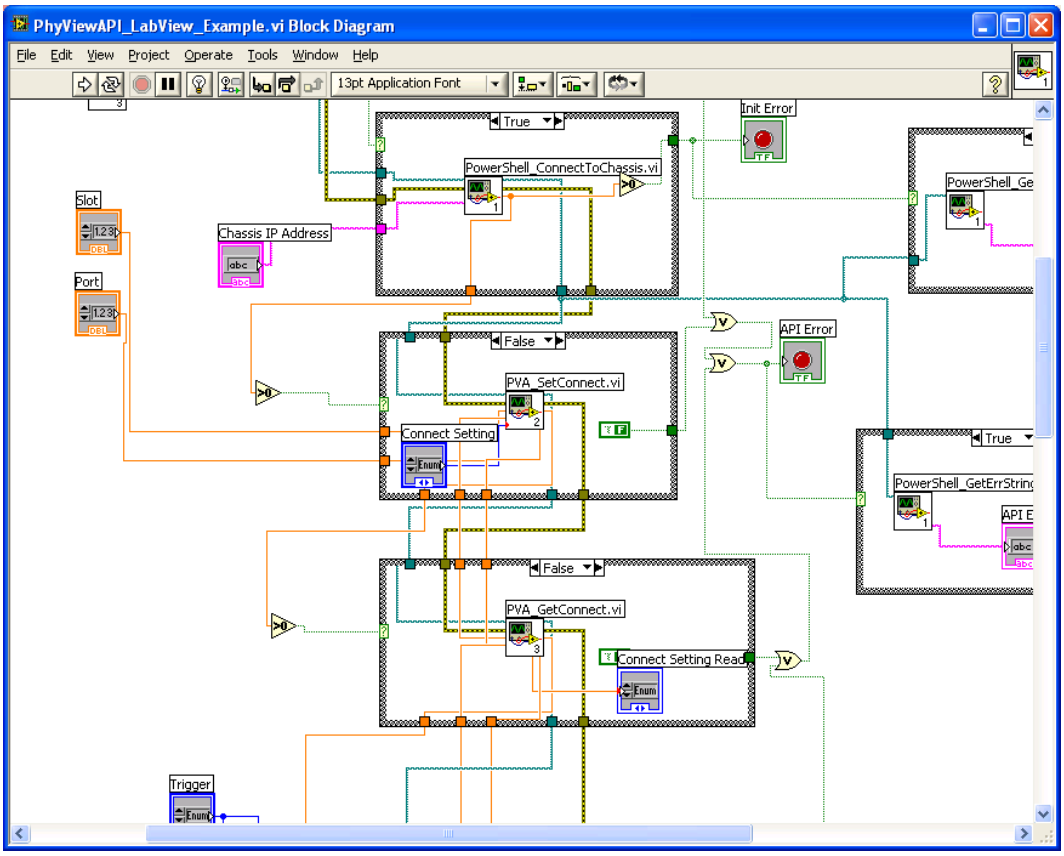

Figure 4.6-3

4.7. LabView - API Issues

The only issue that was discovered during testing was that the PVA_Sequence function cannot be called with the option to produce a CSV file, which subsequently causes the underlying PowerShell PSA Tcl command pva_sequence to programmatically launch Microsoft Excel. A fatal error occurs in LabView if the "-c" option is used with the underlying command. The PVA_Sequence VI does not provide access to the report type parameter, and hard codes the report type to text file.

As described above in section 4.5 **LabView DLL Loading/Unloading Behavior**, there is an issue that has been identified when unloading the DLL when a LabView based program is terminating.

There are no other known issues at this time with regard to LabView, VIs, and the underlying API. The program that was used to test all of the VIs is PhyViewAPI_LabView_TestHarness.vi.

5. PhyView API Example Code

Example programs are provided that illustrate how to call various API functions. The sample program connects to a chassis, performs an inventory, configures a test port, and performs various measurements with that port. API functions that perform each of the three fundamental interactions with a PhyView Analyzer: 1) configuration, 2) settings query, and 3) stat query are used in the example.

5.1. C Code

A C program example named **`PhyViewAPI_Example.c`** is included in the API Library .ZIP archive, under the directory \Example_Code\PhyViewAPI\C_Program. The associated project was created using Microsoft Visual Studio 2005.

5.2. Visual Basic .NET Code

A Visual Basic .NET example named **`UsingTheAPI.vb`** is included in the API Library .ZIP archive, under the directory \Example_Code\PhyViewAPI\VB_NET_program\PhyViewAPI_Example. The associated project was created using Microsoft Visual Studio 2005. . Note that this directory also contains the source code module PhyViewAPI.vb, which contains enumerations and declaration statements for each of the API functions.

5.3. C# Code

A C# .NET example named **`UsingTheAPI.cs`** is included in the API Library .ZIP archive, under the directory \Example_Code\PhyViewAPI\ Csharp_program\PhyViewAPI_Example. The associated project was created using Microsoft Visual Studio 2005. Note that this directory also contains the class module PhyViewAPI.cs, which contains enumerations and methods for each of the API functions.

5.4. LabView Code

A LabView example has been provided, which demonstrates how to connect to a PhyView Analyzer (PVA), configure a port, and perform various measurements with that port. The Front Panel view of the LabView example program is shown in Figure 5.4-1.

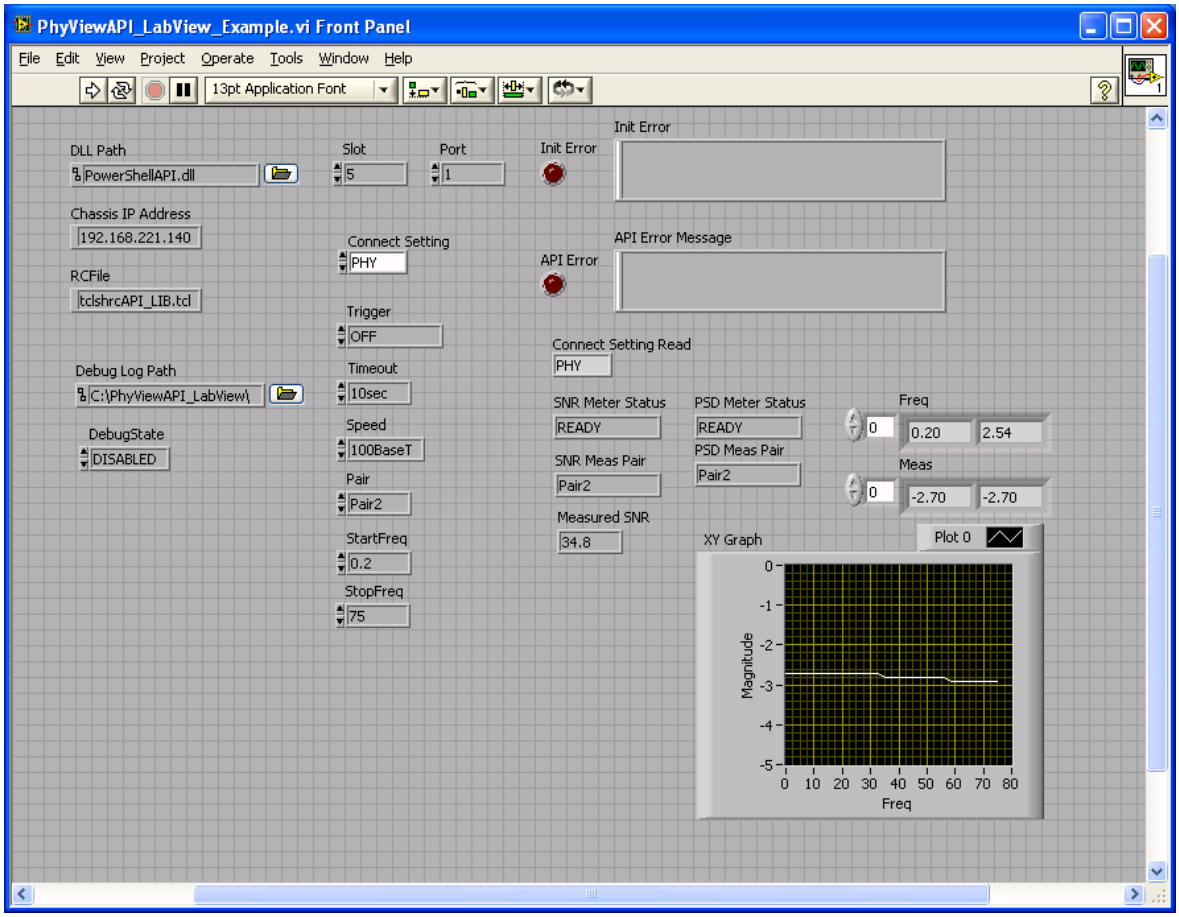

Figure 5.4-1

The API library needs to be loaded, and the initial connection to the PSA established, before any actions can be performed with the instrument. Each VI that encapsulates a function call in the API library requires an input that defines the path to the DLL file that contains that function call. Note that the Path String object located at the upper left of the Front Panel, which contains the path to the DLL in the example directory. The output terminal of this object is connected to the input of the first VI to be executed, as shown in the Block Diagram in Figure 5.4-2.

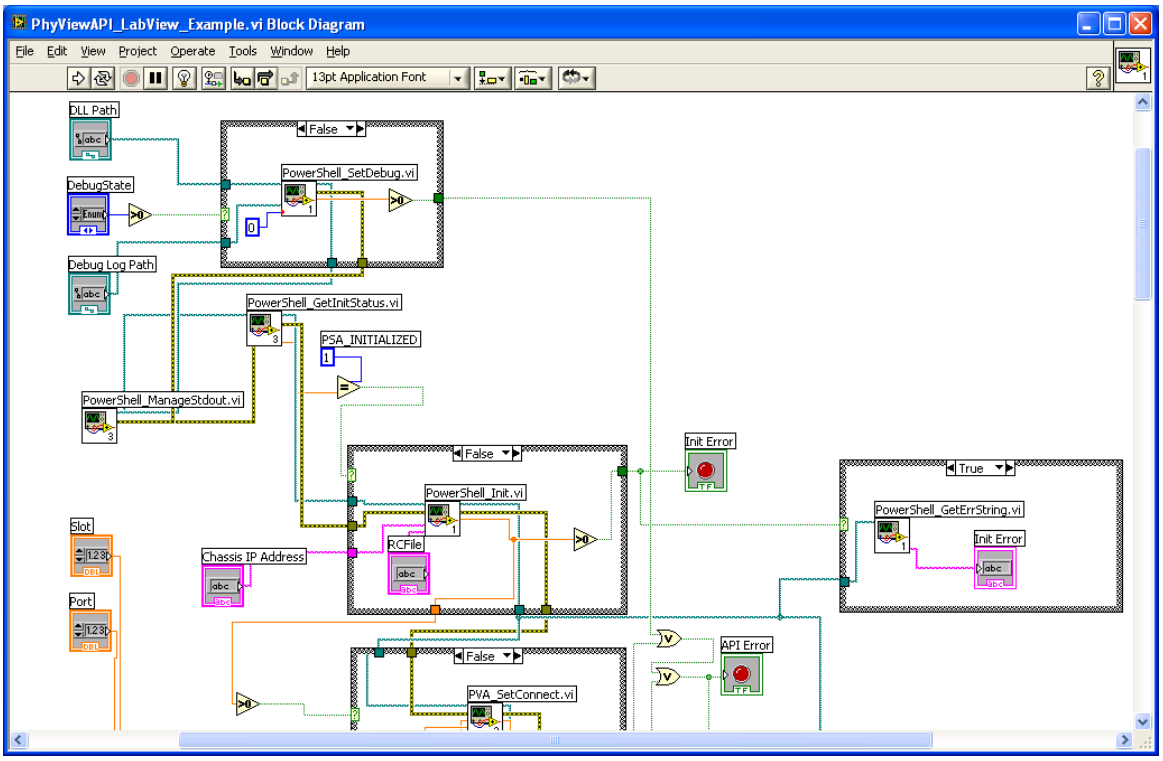

Figure 5.4-2

In this case, the first VI called is PowerShell_SetDebug.vi, which contains a **Call Library Function** node that calls the `PowerShell_SetDebug` function in the API DLL. The purpose of this function is to provide the means to have the API library generate debug output that can be used by Sifos support in the case of any unexpected behavior by the API library. The use of this VI is optional, and it can be omitted if no debug information is required.

The API library is written to support a variety of application environments. As described above, the API library initializes a Tcl interpreter, and loads the PowerShell PSA extensions. The Tcl interpreter supports direct output to a console (also referred to as "`stdout`"). In the case of LabView, there is no `stdout` channel available, and a runtime error will occur unless Tcl is configured to disable the `stdout` channel. This is accomplished by calling the `PowerShell_ManageStdout` function.

The initial connection to the PowerSync Analyzer chassis is performed by the `PowerShell_Init` function. This function requires the IP address of the chassis, and the name of a Runtime Configuration ("RC") file. An example RC file is furnished with the LabView API example – the principal difference between this RC file and the default files used by PowerShell PSA is that this file defines a 0 sec wait time for attempting the connection to the PSA chassis.

NOTE: as described above, once the DLL has been loaded and successfully initialized, the `PowerShell_Init` function cannot be called again for the lifetime of the LabView process that loaded the DLL. The example program contains logic that uses the return value from the function `PowerShell_GetInitStatus` to determine whether or not the API library has been previously initialized. If yes, then the init function can be bypassed. The example program shows an alternative, where the TRUE case includes a call to the `PowerShell_ConnectToChassis` function if the library is detected as being already initialized. This is illustrated in the Block Diagram shown in Figure 5.4-3.

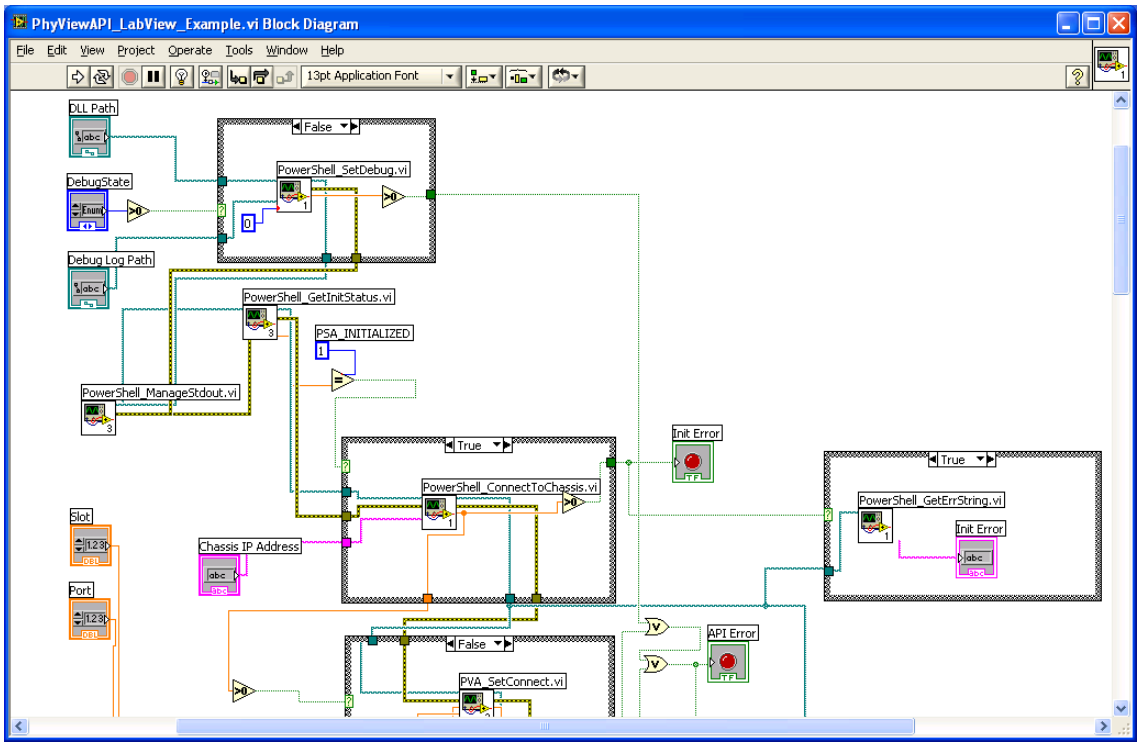

Figure 5.4-3

For any meter that returns a single value (such as SNR, Rx Power), the VI returns one DOUBLE value on an output terminal. For the PSD meter, the VI returns two 1-dimension arrays of DOUBLEs - one containing the list of frequencies that measurements were performed at, and one containing the magnitude data trace data points. This is shown in Figure 5.4-4. Note that this Block Diagram illustrates how the output can be used to build a Cluster, which can then be converted to the 2-dimension array that is needed as the input to an XY Graph object.

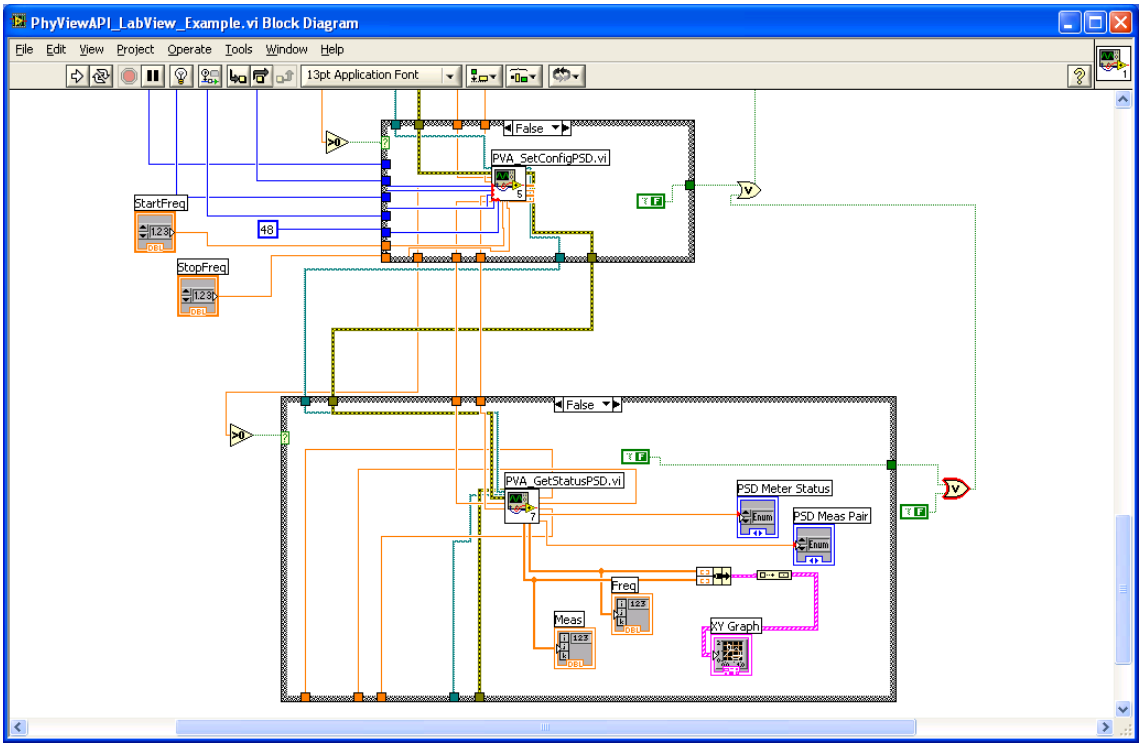

Figure 5.4-4