Sifos Technologies

PowerSync® Device Analyzer

PDA-300

API Library Reference Manual

Version 3.3

Revised October 4, 2012

Copyright © 2012 Sifos Technologies

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

Disclaimer

The information contained in this manual is the property of Sifos Technologies, Inc., and is furnished for use by recipient only for the purpose stated in the Software License Agreement accompanying the user documentation. Except as permitted by such License Agreement, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior written permission of Sifos Technologies, Inc.

Information contained in the user documentation is subject to change without notice and does not represent a commitment on the part of Sifos Technologies, Inc. Sifos Technologies, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in the user documentation.

Table of Contents

1. Introduction

1.1. PDA-300 API Library Introduction

The Sifos Technologies PDA-300 Powered Device Analyzer is a test instrument designed to 1) assure interoperability of a Powered Device (PD) under the IEEE 802.3at standard, and 2) build confidence that PD's are properly specified for purposes of classification and mutual identification with PSE's.

The PDA-300 can operate either as a stand-alone instrument or under control of PC hosted software. Please refer to the **PDA-300 Technical Reference Manual** for detailed information regarding the operation of the instrument and its test capabilities.

This document describes an Application Programming Interface (API) library that allows the instrument to be remotely controlled from a PC over an RS-232 serial connection. This API library is published as a Dynamic Link Library (DLL) for use on Microsoft Windows platforms.

On a Windows platform, any language capable of calling Win32 API functions should be able to use the PDA-300 API library.

The PDA-300 remote control capability effectively provides the means for the front panel softkeys to be 'pressed' programmatically via the serial interface. Front panel softkeys cause the instrument to modify configuration settings, change menu contexts, apply power to a PD, and perform measurements. After a softkey is programmatically selected, the related menu context specific text is output by the instrument via the serial interface. The API library completely encapsulates these menu operations, cycling through the correct menus in the required order to provide a caller with the capability to defining the instrument settings, to enable and disable Vport, and to perform specific measurements by calling functions specific to each of those actions.

For efficiency, the functions in the API library are partitioned to provide the caller with the means of establishing the desired settings (ALT, Polarity, number of Classification Events, and Vport level) before traversing the menus required to modify these settings in the instrument. If the ALT, Polarity, and number of Classification Events settings are not changed, the instrument can remain in a single menu context that allows Vport to turned on and off, the voltage level output to be adjusted, and also allows all of the measurements to be performed. If any one of the ALT, Polarity, and number of Classification Events settings are altered, the instrument will have to be cycled back through the relevant menus to modify these settings. The API library will choose the appropriate action any time a function that is dependent on the configuration settings is called. These functions are the ones that perform measurements, and the one that allows Vport to be turned on.

1.2. System Requirements

For Microsoft Windows

Windows 2000 SP3 or later, through Windows 7
A serial port capable of supporting 9600 baud, 8 bit, no parity, 1 stop bit.

1.3. Installation

The DLL and related files are furnished on the **Sifos PDA-300** CD-ROM, located under the directory \API_files.

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:

1. searches under the current working directory
2. searches under C:\Windows
3. searches under C:\Windows\System32
4. 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.

1.4. Reference Manual Organization

Section 2 of this manual contains the PDA-300 API function definitions, and related enumeration type definitions for any language able to call a standard C function.

Section 3 of this manual contains information related to using the PDA-300 API with different languages.

Section 4 of this manual contains example code showing how to call functions in the PDA-300 API library.

2. PDA-300 API Function Definitions

The PDA-300 API function prototypes and associated enumerations are defined in the file pda300API.h.

As described above, the functions in the API library are partitioned to insure the efficiency in the number of menu traversals required. The function listed in Table 1, Library Settings Only Functions provide the caller with the means of establishing the desired settings (ALT, Polarity, number of Classification Events, and Vport level) before traversing the menus required to modify these settings in the instrument. **Note**: these functions do not perform any I/O with the instrument – the values that they communicate to the API library are stored in local memory, and will be used to configure the instrument settings when one of the functions listed in Table 2 Instrument Operation Functions is called.

Table 1
Library Settings Only Functions

pda300_SetAlt
pda300_GetAlt
pda300_SetMdi
pda300_GetMdi
pda300_SetEvents
pda300_GetEvents
pda300_SetVportLevel
pda300_GetVportLevel

Table 2
Instrument Operation Functions

pda300_MeasRdet
pda300_MeasCdet
pda300_MeasClass
pda300_SetVportState
pda300_GetVportState
pda300_MeasPowerDraw
pda300_AdjustVportLevel
pda300_StartTraceCapture

2.1. Calling Convention

On Microsoft Windows platforms, the calling convention required to call the API functions is `__stdcall`.

2.2. Error Handling

Each of the PDA-300 API functions returns a status value. The various status values are listed in Table 3 Function Return Values.

Table 3
Function Return Values

Integer Value	#define Name
0	PDA300_NO_ERROR
1	PDA300_SERIAL_COMM_ERROR
2	PDA300_INVALID_FW_VERSION
3	PDA300_INVALID_PARAMETER_VALUE
4	PDA300_INVALID_ICLASS
5	PDA300_NULL_POINTER
6	PDA300_FILE_ERROR
7	PDA300_MENU_ERROR
8	PDA300_NOT_IN_LM_MENU
9	PDA300_FAST_SAMPLE_RUNNING
10	PDA300_FAST_SAMPLE_NOT_RUNNING
11	PDA300_TRACE_ERROR
12	PDA300_VPORT_NOT_ENABLED
13	PDA300_TRACE_FILE_FORMAT_ERROR
14	PDA300_TEST_NEVER_COMPLETED

A user developed application program should **always** test the status returned by an API function.

When an API function returns a status other than PDA300_NO_ERROR, the calling application can call the API function pda300_GetErrMessage to retrieve a message string associated with that error.

2.3. Enumerations

```
enum _pda300AltSetting { ALT_A = 1, ALT_B = 2 };

typedef enum _pda300AltSetting pda300AltSetting;

enum _pda300MdiSetting { MDI = 1, MDIX = 2 };

typedef enum _pda300MdiSetting pda300MdiSetting;

enum _pda300ClassEvents { ONE_EVENT = 1, TWO_EVENTS = 2 };

typedef enum _pda300ClassEvents pda300ClassEvents;

enum _pda300VportState { VPORT_OFF = 0, VPORT_ON = 1 };

typedef enum _pda300VportState  pda300VportState;

enum _pda300Quadrants { ONE_QUADRANT = 1, FOUR_QUADRANTS = 2 };

typedef enum _pda300Quadrants pda300Quadrants;
```

2.4. Functions

The more likely error return values are listed for each function. Other return values are possible – refer to Table 3 in the Error Handling section of this document for all possible return values.

2.4.1. pda300_Connect

function: `pda300_Connect`

description: opens a handle to the defined COM port, and attempts to communicate with a PDA-300. If an instrument responds, the firmware version is checked to verify that it will support this API library. If the instrument is compatible with the API library, the front panel is locked out. The hardware version is read and stored, to support decisions related to the sample rate (rates are specified in the PDA-300 Technical Reference Manual).

NOTE: if Vport is ON when this function is called, it will be turned off as the connection is established and the initial command executed.

NOTE: if the library exits abnormally for any reason (for example, an abnormal termination of a program that is using the API library), before the function pda300_Disconnect is called, the instrument's front panel may still be locked out. When the instrument's front panel is locked out, all button presses will be ignored. The easiest way to recover from this state is to cycle power to the instrument. If the instrument will still function correctly with the serial port, you could call pda300_Connect, followed by pda300_Disconnect, which will clear the lockout.

prototype: `int pda300_Connect(char *COMPortName);`

parameters: `char *COMPortName`

For Windows platforms, this should be "COMn", where n is any legal port numbered assigned by the operating system.

COMPortName must contain a NULL terminated string compatible with the C language. For languages other than C/C++, the storage space pointed to should contain an array of characters, with the last character set to NULL (the equivalent to '\0', which is 0x00).

The API library will not be able to open the COM port if another program already has it open. For example, if you have a Hyperterminal window connected to the PDA-300, you will encounter the following error condition when calling the pda300_Connect function:

The function will return PDA300_SERIAL_COMM_ERROR.
The related error message will be: pda300_Connect: CreateFile failed with error 5.

The error 5 indicates that the Windows operating system returned the error ERROR_ACCESS_DENIED.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_NULL_POINTER | PDA300_INVALID_FW_VERSION

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_Connect("COM4");
```

2.4.2. pda300_Disconnect

function: `pda300_Disconnect`

description: disables the lockout of the PDA-300 front panel, and closes the handle to the COM port. This function will return PDA300_SERIAL_COMM_ERROR if a valid COM port connection was not established by a previous call to pda300_Connect.

prototype:
```
int pda300_Disconnect(void);
```

parameters: `none`

The function will return PDA300_SERIAL_COMM_ERROR.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_Disconnect();
```

2.4.3. pda300_GetVersion

function: `pda300_GetVersion`

description: returns the version of the firmware read from the instrument when pda300_Connect was executed. **NOTE**: no I/O is performed when this function is executed.

prototype:
```
float pda300_GetVersion(void);
```

parameters: `none`.

returns: a floating point number, for example, 3.15.

example:
```
float fwVersion;
fwVersion = pda300_GetVersion ();
```

2.4.4. pda300_GetHardwareVer

function: `pda300_GetHarwareVer`

description: returns the version of the hardware read from the instrument when pda300_Connect was executed. **NOTE**: no I/O is performed when this function is executed.

prototype:
```
int pda300_GetHardwareVer(void);
```

parameters: `none`.

returns: an integer number, for example 2.

example:
```
int hwVersion;
hwVersion = pda300_GetHarwareVer ();
```

2.4.5. pda300_SetAlt

function: `pda300_SetAlt`

description: stores the defined ALT setting, to be used when detection or classification measurements are performed, and when Vport is enabled. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

The default ALT setting is ALT_A.

prototype: `int pda300_SetAlt(pda300AltSetting Setting);`

parameters: `pda300AltSetting Setting` - ALT_A | ALT_B

For any language that does not support enumerations, the value (ALT_A = 1, ALT_B = 2) should be passed as a 4-byte (32-bit) integer.

returns: PDA300_NO_ERROR | PDA300_INVALID_PARAMETER_VALUE

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_SetAlt (ALT_A);
```

2.4.6. pda300_GetAlt

function: `pda300_GetAlt`

description: gets the stored ALT setting. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

prototype: `int pda300_GetAlt(pda300AltSetting *Setting);`

parameters: `pda300AltSetting *Setting` - pointer to location to store the ALT setting in. The value stored will be ALT_A | ALT_B (for a language that does not support enumerations ALT_A = 1, ALT_B = 2).

For any language that does not support enumerations, the pointer passed in should refer to a 4-byte (32-bit) integer storage location.

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
pda300AltSetting Setting;
status = pda300_GetAlt (&Setting);
```

2.4.7. pda300_SetMdi

function: `pda300_SetMdi`

description: stores the defined MDI polarity setting, to be used when detection or classification measurements are performed, and when Vport is enabled. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

The default MDI polarity setting is MDI.

prototype: `int pda300_SetMdi(pda300MdiSetting Setting);`

parameters: `pda300MdiSetting Setting` - MDI | MDIX

For any language that does not support enumerations, the value (MDI = 1, MDIX = 2) should be passed as a 4-byte (32-bit) integer.

returns: PDA300_NO_ERROR | PDA300_INVALID_PARAMETER_VALUE

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_SetMdi (MDIX);
```

2.4.8. pda300_GetMdi

function: `pda300_GetMdi`

description: gets the stored MDI polarity setting. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

prototype: `int pda300_GetMdi(pda300MdiSetting *Setting);`

parameters: `pda300MdiSetting *Setting` - pointer to location to store the MDI setting in. The value stored will be MDI | MDIX (for a language that does not support enumerations MDI = 1, MDIX = 2).

For any language that does not support enumerations, the pointer passed in should refer to a 4-byte (32-bit) integer storage location.

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
 pda300MdiSetting Setting;
status = pda300_GetMdi (&Setting);
```

2.4.9. pda300_SetEvents

function: `pda300_SetEvents`

description: stores the defined number of Classification events setting, to be used when classification measurements are performed, and when Vport is enabled. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

The default Events setting is TWO_EVENTS.

prototype: `int pda300_SetEvents(pda300ClassEvents Setting);`

parameters: `pda300ClassEvents Setting` - ONE_EVENT | TWO_EVENTS

For any language that does not support enumerations, the value (ONE_EVENT = 1, TWO_EVENTS = 2) should be passed as a 4-byte (32-bit) integer.

returns: PDA300_NO_ERROR | PDA300_INVALID_PARAMETER_VALUE

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_ SetEvents (TWO_EVENTS);
```

2.4.10. pda300_GetEvents

function: `pda300_GetEvents`

description: gets the stored number of Classification events. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

prototype: `int pda300_GetEvents(pda300ClassEvents *Setting);`

parameters: `pda300ClassEvents Setting` - pointer to location to store the ClassEvents setting in. The value stored will be ONE_EVENT | TWO_EVENTS (for a language that does not support enumerations ONE_EVENT = 1, TWO_EVENTS = 2).

For any language that does not support enumerations, the pointer passed in should refer to a 4-byte (32-bit) integer storage location.

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
pda300ClassEvents Setting;
status = pda300_GetEvents (&Setting);
```

2.4.11. pda300_SetVportLevel

function: `pda300_SetVportLevel`

description: stores the defined voltage setting, to be used as the voltage level when Vport is enabled. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function. Also, be aware that this function will not cause the voltage to change once the port is powered on.

The voltage level can be changed once the port is powered on by calling the function pda300_AdjustVportLevel.

The default Vport level is 54V.

prototype:
```
int pda300_SetVportLevel(int Voltage);
```

parameters: `int Voltage` - 28…57

For languages other than C/C++, the Voltage must be passed as a 4-byte (32-bit) value.

returns: PDA300_NO_ERROR | PDA300_INVALID_PARAMETER_VALUE

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_SetVportLevel (48);
```

2.4.12. pda300_GetVportLevel

function: `pda300_GetVportLevel`

description: gets the stored voltage setting. **NOTE**: no I/O is performed when this function is executed – this is a Library Settings Only function.

prototype:
```
int pda300_GetVportLevel(int *Voltage);
```

parameters: `int Voltage` - pointer to location to store the Vport setting in.

For languages other than C/C++, the storage space pointed to should represent a 4 byte (32-bit) integer.

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
int Voltage;
status = pda300_GetVportLevel (&Voltage);
```

2.4.13. pda300_MeasRdet

function: `pda300_MeasRdet`

description: measures the detection resistance R_{det} exhibited by the connected PD.

NOTE: this function has a side effect related to the state of Vport. This measurement can only be performed when the PDA-300 is **not** applying voltage to the PD. If the PDA-300 is applying a voltage when this function is called, the voltage output is turned off, and the measurement performed. Vport is **not** turned on again after the measurement is performed. The user would have to call pda300_SetVportState to turn Vport back on.

prototype: `int pda300_MeasRdet (float *Rdet);`

parameters: `float *Rdet` - pointer to a location to store the measured R_{det} in.

For languages other than C/C++, the storage space pointed to should represent a 4 byte (IEEE 32-bit) floating point number. For example, in Visual Basic, the appropriate type is "single".

The value stored in Rdet is resistance, in units of kohms. The API library parses the value reported by the instrument. The PDA-300 will report values in the following ways:

For a valid detection resistive signature, a value such as 24.00k.
For an open circuit, the value 99.99k.
With the wire pairs shorted together, the value 00.00k.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
float Rdet;
status = pda300_MeasRdet (&Rdet);
```

2.4.14. pda300_MeasCdet

function: `pda300_MeasCdet`

description: measures the detection capacitance C_{det} exhibited by the connected PD.

NOTE: this function has a side effect related to the state of Vport. This measurement can only be performed when the PDA-300 is not applying voltage to the PD. If the PDA-300 is applying a voltage when this function is called, the voltage output is turned off, and the measurement performed. Vport is **not** turned on again after the measurement is performed. The user would have to call pda300_SetVportState to turn Vport back on.

prototype: `int pda300_MeasCdet (float *Cdet);`

parameters: `float *Cdet` - pointer to a location to store the measured C_{det} in.

For languages other than C/C++, the storage space pointed to should represent a 4 byte (IEEE 32-bit) floating point number. For example, in Visual Basic, the appropriate type is "single".

The value stored in Cdet is resistance, in units of micro farads. The API library parses the value reported by the instrument. The PDA-300 will report values in the following ways:

For a valid detection capacitive signature, a value such as 0.109uF.
For an open circuit, the value 0.004uF.
With the wire pairs shorted together, the value 99.999uF.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
float Cdet;
status = pda300_MeasCdet (&Cdet);
```

2.4.15. pda300_MeasClass

function: `pda300_MeasClass`

description: measures the classification current drawn by the connected PD, and determines the related 802.3at defined class. NOTE: if the class current is invalid, the Class value returned is 0.

NOTE: this function has a side effect related to the state of Vport. This measurement can only be performed when the PDA-300 is not applying voltage to the PD. If the PDA-300 is applying a voltage when this function is called, the voltage output is turned off, and the measurement performed. Vport is **not** turned on again after the measurement is performed. The user would have to call pda300_SetVportState to turn Vport back on.

prototype: `int pda300_MeasClass (float *Iclass, int *Class);`

parameters: `float *Iclass` - pointer to a location to store the measured class current in.
`int *Class` - pointer to a location to store the Class number in.

For languages other than C/C++, the storage space pointed to for Iclass should represent a 4 byte (IEEE 32-bit) floating point number. For example, in Visual Basic, the appropriate type is “single”.

The pointer passed in for Class should refer to a 4-byte (32-bit) integer storage location.

The value stored in Iclass is current, in units of mA.
The value stored in Class is an integer 0 | 1 | 2 | 3 | 4

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_INVALID_ICLASS | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
float Iclass;
int Class, Status;
status = pda300_MeasClass (&Iclass, &Class);
```

2.4.16. pda300_SetVportState

function: `pda300_SetVportState`

description: sets the state of Vport to the caller defined state – ON or OFF. Classification, using the specified number of event, will be performed prior to Vport being turned ON.

prototype: `int pda300_SetVportState(pda300VportState State);`

parameters: `pda300VportState State` - VPORT_OFF | VPORT_ON

For any language that does not support enumerations, the value (VPORT_OFF = 0, VPORT_ON = 1) should be passed as a 4-byte (32-bit) integer.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_INVALID_PARAMETER_VALUE

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_SetVportState (VPORT_ON);
```

2.4.17. pda300_GetVportState

function: `pda300_GetVportState`

description: gets the state of Vport – ON or OFF.

prototype: `int pda300_GetVportState(pda300VportState *State);`

parameters: `pda300VportState *State` - pointer to location to store the Vport state in. The value stored will be VPORT_OFF | VPORT_ON (for a language that does not support enumerations VPORT_OFF = 0, VPORT_ON = 1).

For any language that does not support enumerations, the pointer passed in should refer to a 4-byte (32-bit) integer storage location.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
pda300VportState State;
status = pda300_GetVportState (&State);
```

2.4.18. pda300_AdjustVportLevel

function: `pda300_AdjustVportLevel`

description: changes the active Vport level to the caller defined value. NOTE: Unlike the function defined above (pda300_SetVportLevel), this function DOES perform instrument I/O.

NOTE: Vport must be turned ON before this function is called. This is accomplished by calling the function pda300_SetVportState.

NOTE: this function has a side effect related to the stored Vport level. The Voltage level passed to this function becomes the new level stored by the library, and will be the initial level used if pda300_SetVportState is called to turn the port voltage ON without a preceding call to pda300_SetVportLevel.

The roles of this function and pda300_SetVportLevel were partitioned based on menu traversal efficiency, as described above.

prototype: `int pda300_AdjustVportLevel (int Voltage);`

parameters: `int Voltage` - 28…57

For languages other than C/C++, the Voltage must be passed as a 4-byte (32-bit) value.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR |
PDA300_INVALID_PARAMETER_VALUE | PDA300_VPORT_NOT_ENABLED

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_AdjustVportLevel(51);
```

2.4.19. pda300_MeasPowerDraw

function: `pda300_MeasPowerDraw`

description: measures the current drawn by the powered PD, and calculates the Power consumption using the measured current and Vport value.

prototype: `int pda300_MeasPowerDraw (float *Iload, float *Pload);`

parameters: `float *Iload` - pointer to a location to store the measured current in.
`float *Pload` - pointer to a location to store the calculated power in.

For languages other than C/C++, the storage space pointed to for Iload and Pload should each represent a 4 byte (IEEE 32-bit) floating point number. For example, in Visual Basic, the appropriate type is "single".

The value stored in Iload is current, in units of mA.
The value stored in Pload is power, in units of Watts.

returns: PDA300_NO_ERROR | PDA300_SERIAL_COMM_ERROR | PDA300_INVALID_ICLASS | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
float Iload;
float Pload;
status = pda300_MeasPowerDraw (&Iload, &Pload);
```

2.4.20. pda300_StartTraceCapture

function: `pda300_StartTraceCapture`

description: directs the API library to open the indicated file for write operations, and enable a fast data sample streaming mode on the PDA-300. The calling process must have write permission for the defined path.

Prior to enabling fast data sample streaming mode, the API library starts a thread, a code module which executes in parallel with the existing process that called the pda300_StartTraceCapture function. As the data being transmitted by the instrument is read by the thread, it is stored in the indicated file.

NOTE: the data is transmitted by the instrument at the rate of one sample every 20ms, or 50 samples/sec (hardware version 1); 18.52ms, 54 samples/sec (hardware version 2). Each sample is comprised of a 16-bit binary value, and a delimiter character (a comma). The resulting data storage requirement is 150 bytes/sec. The file will grow at a rate of 9000 bytes/min. The caller needs to insure that they have adequate disk space to accommodate a trace of the time length required. The amount of available space needs to be at least 11x the binary file size, as the file will translated from binary to a CSV formatted ASCII file when the trace is stopped.

For example, a trace running for an hour should produce a binary data file that is 540000 bytes in size.

NOTE: the file defined by Path is opened with mode = write. This will create a new file if it does not already exist, and will destroy the contents of a file that already exists.

NOTE: when the fast data sample streaming mode is enabled on the PDA-300, no other functions that perform I/O can be called. The instrument is *very* busy capturing current draw samples, and transmitting them via the serial interface. If any of those functions are called while the fast data sample streaming mode is enabled, an error will be returned.

The only function that can be called once fast data sample streaming mode is enabled is pda300_StopTraceCapture.

NOTE: When using an instrument running firmware versions < 3.15, Vport must be turned ON before starting the trace capture (an error will be returned if Vport is OFF). If the instrument is running firmware version 3.15 (or later), Vport will be turned on by the firmware if it is not already on when the trace is started. This allows any initial transient power consumption to be observed at the beginning of the trace. In either case, classification, using the defined number of events, is performed prior to Vport being turned on.

prototype: `int pda300_StartTraceCapture (char *Path);`

parameters: `char *Path` - pointer to char array that contains the path to a file to write the trace to.

Path must contain a NULL terminated string compatible with the C language. For languages other than C/C++, the storage space pointed to should contain an array of characters, with the last character set to NULL (the equivalent to '\0', which is 0x00).

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER | PDA300_FILE_ERROR | PDA300_TRACE_ERROR | PDA300_FAST_SAMPLE_RUNNING

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_StartTraceCapture ("/temp/pd_trace.csv");
```

2.4.21. pda300_StopTraceCapture

function: `pda300_StopTraceCapture`

description: directs the API library to terminate the fast data sample streaming mode on the PDA-300. Once the trace has terminated, the binary sample data is translated from binary format to a CSV formatted ASCII file. The file that the trace will end up in is the one defined by the Path argument passed to pda300_StartTraceCapture. A file named *{path}*BIN will remain when the trace processing has completed. This file may be deleted at your convenience (they are retained at trace completion for possible debug support).

NOTE: when the fast data sample streaming mode is enabled on the PDA-300, the only function that can be called once fast data sample streaming mode is enabled is pda300_StopTraceCapture. If pda300_StopTraceCapture is called and fast data sample streaming mode is not enabled, an error will be returned.

The CSV file will contain four columns of data:

1. Time - the tine at which each sample was acquired, in units of seconds
2. Iload - the measured current, in units of amps
3. Pload - the calculated power at this specific sample point (Iload * Vport), in units of Watts
4. Pload_avg - the running average power, calculated over the last 1 second, in units of Watts

NOTE: the Pload_avg value is listed as 0 until one second of data has been accumulated. From that point on, the running average power is calculated for each point. This corresponds to the average load power as defined in the 802.3at specification.

Example data:

Time(sec),Iload(A),Pload(W),Pload_avg(W)
0.00,0.0007,0.0,0.0
0.02,0.0203,1.0,0.0
0.04,0.0201,1.0,0.0
0.06,0.0201,1.0,0.0
0.08,0.0202,1.0,0.0
0.10,0.0207,1.0,0.0
• • •
0.94,0.0203,1.0,0.0
0.96,0.0203,1.0,0.0
0.98,0.0201,1.0,0.9 ← Pload_avg data starts here
1.00,0.0204,1.0,0.9
1.02,0.0201,1.0,0.9
1.04,0.0203,1.0,0.9

prototype: `int pda300_StopTraceCapture (void);`

parameters: none

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER | PDA300_FILE_ERROR | PDA300_TRACE_ERROR | PDA300_FAST_SAMPLE_NOT_RUNNING

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_StopTraceCapture ( );
```

2.4.22. pda300_Run802dot3ATtest

function: `pda300_Run802dot3ATtest`

description: initiates the built-in 802.3at test on the PDA-300, and stores the results in the indicated file when the test completes. The calling process must have write permission for the defined path.

The test results are stored as a CSV formatted ASCII file. The first row contains column labels, and the remaining rows each start with a definition of the topology that row contains test results for.

Example data:

```
Quadrant,Rdet,Status,Cdet,Status,Iclass,Status,Class,Status,Type,Von,Status,Voff,Status,Ei
nr,Status,Power1,Status,Pk_pwr1,Status,Imax1,Status,Imin1,Status,Iavg1,Imark,Status,Power2
,Status,Pk_pwr2,Status,Pinit2,Status,Imax2,Status,Imin2,Status,Iavg2
Alt-A MDI,23.86k,P,0.106uF,P,40.0mA,P,4,P,2,39.2V,P,31.9V,P,0.08W-
s,P,0.98W,P,1.01W,P,20.9mA,-,20.3mA,P,20.6mA,0.68mA,P,1.04W,P,1.06W,P,1.06W,P,19.8mA,-
,19.1mA,P,19.4mA,

Alt-A MDI-X,24.02k,P,0.106uF,P,40.0mA,P,4,P,2,

Alt-B MDI,24.00k,P,0.104uF,P,40.0mA,P,4,P,2,

Alt-B MDI-X,24.00k,P,0.106uF,P,40.0mA,P,4,P,2,
```

prototype: `int pda300_Run802dot3ATtest (pda300Quadrants Quadrants, char *Path);`

parameters: `pda300Quadrants Quadrants` - defines the number of parameters to test.

`ONE_QUADRANT` performs all tests for Alt-A MDI, and unpowered tests for all other topologies
`FOUR_QUADRANTS` performs all tests for all topologies

`char *Path` - pointer to char array that contains the path to a file to write the test results to.

Path must contain a NULL terminated string compatible with the C language. For languages other than C/C++, the storage space pointed to should contain an array of characters, with the last character set to NULL (the equivalent to '\0', which is 0x00).

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER | PDA300_FILE_ERROR | PDA300_FAST_SAMPLE_RUNNING | PDA300_TEST_NEVER_COMPLETED

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
    status = pda300_Run802dot3ATtest (ONE_QUADRANT, "/temp/oneQuad.csv");
```

2.4.23. pda300_GetErrMessage

function: `pda300_GetErrMessage`

description: gets the error message string. No I/O is performed when this function is executed.

prototype: `int pda300_GetErrMessage (char *Buf, int LenBuf);`

parameters: `char *Buf` - pointer to location to store the error message string in.
`int LenBuf` - length of char array being passed in

For languages other than C/C++, the storage space pointed to must represent an array of 8-bit characters that the API function is allowed to write to.

NOTE: is it *very* important for LenBuf to accurately represent the length of the character array. There is no protection from overrunning a memory location in the C language without knowledge of the amount of space that has been allocated. If the length is not specific accurately, unexpected behavior, such as a fatal error, may occur in the calling process.

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

example:
```
#define LENGTH_ERR_MSG 256
int status;
char ErrMsg[LENGTH_ERR_MSG];
status = pda300_GetErrMessage (&ErrMsg[0], LENGTH_ERR_MSG);
```

2.4.24. pda300_SetDebugPath

function: `pda300_SetDebugPath`

description: directs the API library to open the indicated file for write operations. No instrument I/O is performed when this function is executed. The calling process must have write permission for the defined path.

Debug is disabled by default when the API library is initially loaded. Debug is enabled by passing in a valid path to a file to write. Debug can be disabled by passing in a NULL.

NOTE: the file defined by Path is opened with mode = write. This will create a new file if it does not already exist, and will destroy the contents of a file that already exists.

The debug mechanism is primarily intended for use to provide Sifos with diagnostic information in the case where unexpected behavior is encountered by the user.

prototype: `int pda300_SetDebugPath (char *Path);`

parameters: `char *Path -` pointer to char array that contains the path to a file to write debug information to.

Path must contain a NULL terminated string compatible with the C language. For languages other than C/C++, the storage space pointed to should contain an array of characters, with the last character set to NULL (the equivalent to '\0', which is 0x00).

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER | PDA300_FILE_ERROR

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_SetDebugPath ("/temp/debug.txt");
```

2.4.25. pda300_DebugFileWrite

function: `pda300_DebugFileWrite`

description: writes the string passed in Buf to the debug file. A Newline (0x0A) character will be appended to the end of the string as it is written to the debug file.

NOTE: this function does nothing of a debug file is not currently open.

prototype: `int pda300_DebugFileWrite (char *Buf);`

parameters: `char *Buf` - pointer to char array that contains a string to be written to the debug file.

Buf must contain a NULL terminated string compatible with the C language. For languages other than C/C++, the storage space pointed to should contain an array of characters, with the last character set to NULL (the equivalent to '\0', which is 0x00).

returns: PDA300_NO_ERROR | PDA300_NULL_POINTER

If an error is returned, the function pda300_GetErrMessage can be called to retrieve a related error message.

example:
```
int status;
status = pda300_DebugFileWrite ("Measuring power consumption.");
```

2.4.26. pda300_SetTimeouts

function: `pda300_SetTimeouts`

description: allows external definition of the timeouts used while performing serial I/O. The API library uses default values that should provide the correct operation (which they have on all test PCs used to-date). The ability to set these values has been exposed to provide the means to adjust these values in the unexpected case where a host PC runs very slow and the API functions experience timeouts.

prototype: `int pda300_SetTimeouts (int stdTimeout, int measTimeout);`

parameters: `int stdTimeout` - timeout used when changing menu contexts, value in msec. Value must be > 0.
`int measTimeout` - timeout used when performing measurements, value in msec. Value must be > 0.

returns: PDA300_NO_ERROR | PDA300_INVALID_PARAMETER_VALUE

example:
```
int status;
status = pda300_SetTimeout (1500, 4500);
```

3. PDA-300 API Library Definitions for Other Languages

The functions in the library pda300API.dll and associated enumerations are defined for Visual Basic 6 in the source module pda300API.bas.

3.1. Visual Basic 6

3.1.1. Type Differences

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

API Type	VB6 Type
float	Single
int	Long

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.

3.1.2. Enumerations

VB6 supports enumerations. The enumerations used by the API library are defined in the pda300API.bas file.

3.1.3. Service Functions

The service function TrimAPIString provides the means to trim any unused space off of the end of a fixed length string containing the value passed back by a pda300API function.

```
Public Function TrimAPIString(sBufr As String)
```

3.1.4. VB6 Compatible Declarations

Visual Basic 6 compatible declarations are located in the source module pda300API.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 pda300_Connect Lib "pda300API.dll" (ByVal COMPortName As
String) As Long
```

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

```
Declare Function pda300_GetErrMessage Lib "pda300API.dll" (ByVal Buf As String,
ByVal LenBuf As Long) As Long
```

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

```
Declare Function pda300_SetAlt Lib "pda300API.dll" (ByVal Setting As
pda300AltSetting) As Long
```

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

```
Declare Function pda300_GetAlt Lib "pda300API.dll" (ByRef Setting As
pda300AltSetting) As Long
```

3.2. Visual Basic .NET

3.2.1. Type Differences

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

API Type	VB .NET Type
float	Single
int	Integer

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(LENGTH_PDA300_ERROR_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.

3.2.2. Enumerations

VB .NET supports enumerations. The enumerations used by the API library are defined in the pda300API.vb file.

3.2.3. Service Functions

None required.

3.2.4. VB .NET Compatible Declarations

Visual Basic .NET compatible declarations are located in the source module pda300API.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 strings allocated in VB .NET using the System.Text.StringBuilder object. 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 pda300_Connect Lib "pda300API.dll" (ByVal COMPortName As
System.Text.StringBuilder) As Integer
```

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

```
Declare Function pda300_GetErrMessage Lib "pda300API.dll" (ByVal Buf As
System.Text.StringBuilder, ByVal LenBuf As Integer) As Integer
```

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

```
Declare Function pda300_SetAlt Lib "pda300API.dll" (ByVal Setting As
pda300AltSetting) As Integer
```

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

```
Declare Function pda300_GetAlt Lib "pda300API.dll" (ByRef Setting As
pda300AltSetting) As Integer
```

3.3. LabView

3.3.1. Type Differences

There are type differences in LabView that you will need to keep in mind, when defining API function pass parameters on the **Parameters** tab of the **Call Library Function** Configure… dialog:

API Type	LabView Type	LabView Data Type	Pass
float	Numeric	4-byte Single	Pointer to Value (for meas function)
int	Numeric	Signed 32-bit Integer	Value (for set function) Pointer to Value (for get function)

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. At the time this section of the manual was modified, the only string that the API library writes to is the one that is passed with the call pda300_GetErrMessage. The size to use for the LabView string is defined by the macro LENGTH_PDA300_ERROR_BUF, located in the header file pda300API.h.

3.3.2. 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 pda300API.h.

3.3.3. LabView VIs

LabView VIs for each of the API functions are located under the path: `\API_Files\LabView` on the **Sifos PDA-300** CD-ROM. Prior to September, 2012, the original VIs did not include Error In or Error Out clusters, and were used in a flat sequence in an example program. These are provided in the directory `\API_Files\LabView\Original_Flat_Sequence_form`.

The VIs were re-worked to include Error In and Error Out clusters, which allow for easier inclusion in a LabView program, providing the necessary sequence control to insure the order of execution. The new form VIs are provided in the directory `\API_Files\LabView\Error_In_Error_Out_form`.

NOTE: these VIs were created with LabView 8.20.

4. PDA-300 API Example Code

4.1. C Code Example

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//
// pda300API_example.c : illustrates how to use the PDA-300 API Library
//
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//
// Console output from the example program
//
//ALT is set to ALT-B
//Polarity is set to MDI-X
//Classification will use TWO Events
//Vport level is set to 54 V
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Performing unpowered measurements
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Rdet: 24.049999 k is in range
//Cdet: 0.106000 uF is in range
//PD Class is: 4  measured Iclass: 40.000000 mA
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Applying Power
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Vport is ON
//With Vport 54 V, PD is drawing: 19.600000 mA and consuming: 1.050000 W
//Reducing Vport
//With Vport 46 V, PD is drawing: 21.700001 mA and consuming: 0.990000 W
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Removing Power, changing Alt and Polarity
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Performing unpowered measurements
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Rdet: 23.940001 k is in range
//Cdet: 0.112000 uF is in range
//PD Class is: 4  measured Iclass: 40.099998 mA
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Applying Power
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Vport is ON
//With Vport 46 V, PD is drawing: 21.700001 mA and consuming: 0.990000 W
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Performing a 10 second trace capture
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Trace file stored in /temp/pda300_trace.csv
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Running the built-in 802.3at test - one quadrant
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//One quadrant 802.3at test results stored in /temp/oneQuadrantResults.csv
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Running the built-in 802.3at test - four quadrants
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Four quadrant 802.3at test results stored in /temp/fourQuadrantResults.csv
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
//Done!
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <windows.h>
#include <stdarg.h>
#include <stdio.h>
#include <time.h>
#include "pda300API.h"

void checkRtn (int iRtn);

int main(int argc, char* argv[])
{
	int iRtn;
	pda300AltSetting eAlt;
	pda300MdiSetting eMdi;
	pda300ClassEvents eEvents;
	int iVport = 0;
	pda300VportState eVportState;
	// NOTE: change the COM port to the one that the instrument is connected to
	//	on your system.
	char COMPortName[10] = { "COM1" };
	float Rdet;
	float Cdet;
	float Iclass;
	int Class;
	float Iload;
	float Pload;
	time_t startTime;
	time_t currentTime;
	// NOTE: change these paths to one that exists on your system.
	char *pszDbgFilePath = "/temp/debug_output.txt";
	char *pszTraceFilePath = "/temp/pda300_trace.csv";
	char *pszOneQuadFilePath = "/temp/oneQuadrantResults.csv";
	char *pszFourQuadFilePath = "/temp/fourQuadrantResults.csv";

	// The call to pda300_SetDebugPath can be uncommented if you want to
	// produce a debug file.
	//iRtn = pda300_SetDebugPath(pszDbgFilePath);
	//checkRtn(iRtn);

	// Open a connection to the instrument, locking out the front panel
	iRtn = pda300_Connect(COMPortName);
	checkRtn(iRtn);

	if (iRtn != PDA300_NO_ERROR)
	{
		// The test program should exit at this point, or
		// indicate an error to the operator, since we
		// cannot communicate with the instrument.
	}

	// Define the ALT setting
	iRtn = pda300_SetAlt(ALT_B);
	checkRtn(iRtn);

	// Interrogate the library to check the ALT setting
	iRtn = pda300_GetAlt(&eAlt);
	checkRtn(iRtn);

	if (eAlt == ALT_A)
		printf ("ALT is set to ALT-A\n");
```

```
else
            printf ("ALT is set to ALT-B\n");

// Define the polarity setting
iRtn = pda300_SetMdi(MDIX);
checkRtn(iRtn);

// Interrogate the library to check the polarity setting
iRtn = pda300_GetMdi(&eMdi);
checkRtn(iRtn);

if (eMdi == MDI)
            printf ("Polarity is set to MDI\n");
else
            printf ("Polarity is set to MDI-X\n");

// Define the number of event to use during Classification
iRtn = pda300_SetEvents(TWO_EVENTS);
checkRtn(iRtn);

// Interrogate the library to check the number of events setting
iRtn = pda300_GetEvents(&eEvents);
checkRtn(iRtn);

if (eEvents == ONE_EVENT)
            printf ("Classification will use ONE Event\n");
else
            printf ("Classification will use TWO Events\n");

// Define the Vport voltage level when power is applied
iRtn = pda300_SetVportLevel(54);
checkRtn(iRtn);

// Interrogate the library to check the voltage level setting
iRtn = pda300_GetVportLevel(&iVport);
checkRtn(iRtn);

printf("Vport level is set to %d V\n", iVport);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Performing unpowered measurements\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

// Measure Rdet
iRtn = pda300_MeasRdet (&Rdet);
checkRtn(iRtn);

if (Rdet < 23.2 || Rdet > 24.8)
            printf("Rdet: %f k is out of range\n", Rdet);
else
            printf("Rdet: %f k is in range\n", Rdet);


// A testplan would probably decide not to proceed with
// powering on the PD, if an irrational Rdet is measured.



// Measure Cdet
iRtn = pda300_MeasCdet (&Cdet);
checkRtn(iRtn);

if (Cdet < 0.09 || Cdet > 0.15)
            printf("Cdet: %f uF is out of range\n", Cdet);
```

```
else
        printf("Cdet: %f uF is in range\n", Cdet);

// Determine what Class the PD represents itself as
iRtn = pda300_MeasClass (&Iclass, &Class);
checkRtn(iRtn);

if (iRtn == PDA300_INVALID_ICLASS)
        printf("Invalid Iclass: %f mA, cannot determine PD Class\n", Iclass);
else
        printf("PD Class is: %d  measured Iclass: %f mA\n", Class, Iclass);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Applying Power\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

// Apply power
iRtn = pda300_SetVportState (VPORT_ON);
checkRtn(iRtn);

// Interrogate the library to check the state of Vport
iRtn = pda300_GetVportState (&eVportState);
checkRtn(iRtn);

if (eVportState == VPORT_ON)
        printf("Vport is ON\n");
else
        printf("Vport is OFF\n");

// Measure the current demand and power consumed by the PD
iRtn = pda300_MeasPowerDraw (&Iload, &Pload);
checkRtn(iRtn);

printf("With Vport %d V, PD is drawing: %f mA and consuming: %f W\n", iVport, Iload, Pload);

printf("Reducing Vport\n");

// Reduce Vport to 46V
iRtn = pda300_AdjustVportLevel(46);
checkRtn(iRtn);

// Interrogate the library to check the voltage level setting
iRtn = pda300_GetVportLevel(&iVport);
checkRtn(iRtn);

// Measure the current demand and power consumed by the PD
iRtn = pda300_MeasPowerDraw (&Iload, &Pload);
checkRtn(iRtn);

printf("With Vport %d V, PD is drawing: %f mA and consuming: %f W\n", iVport, Iload, Pload);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Removing Power, changing Alt and Polarity\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

// Remove power
iRtn = pda300_SetVportState (VPORT_OFF);
checkRtn(iRtn);

// Change settings
iRtn = pda300_SetAlt(ALT_A);
checkRtn(iRtn);

iRtn = pda300_SetMdi(MDI);
```

```
checkRtn(iRtn);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Performing unpowered measurements\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

// Measure Rdet
iRtn = pda300_MeasRdet (&Rdet);
checkRtn(iRtn);

if (Rdet < 23.2 || Rdet > 24.8)
        printf("Rdet: %f k is out of range\n", Rdet);
else
        printf("Rdet: %f k is in range\n", Rdet);

// Measure Cdet
iRtn = pda300_MeasCdet (&Cdet);
checkRtn(iRtn);

if (Cdet < 0.09 || Cdet > 0.15)
        printf("Cdet: %f uF is out of range\n", Cdet);
else
        printf("Cdet: %f uF is in range\n", Cdet);

// Determine what Class the PD represents itself as
iRtn = pda300_MeasClass (&Iclass, &Class);
checkRtn(iRtn);

if (iRtn == PDA300_INVALID_ICLASS)
        printf("Invalid Iclass: %f mA, cannot determine PD Class\n", Iclass);
else
        printf("PD Class is: %d  measured Iclass: %f mA\n", Class, Iclass);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Applying Power\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

// Apply power...NOTE that the Vport level will be the one that was
// last defined.
iRtn = pda300_SetVportState (VPORT_ON);
checkRtn(iRtn);

// Interrogate the library to check the state of Vport
iRtn = pda300_GetVportState (&eVportState);
checkRtn(iRtn);

if (eVportState == VPORT_ON)
        printf("Vport is ON\n");
else
        printf("Vport is OFF\n");

// Interrogate the library to check the voltage level setting
iRtn = pda300_GetVportLevel(&iVport);
checkRtn(iRtn);

// Measure the current demand and power consumed by the PD
iRtn = pda300_MeasPowerDraw (&Iload, &Pload);
checkRtn(iRtn);

printf("With Vport %d V, PD is drawing: %f mA and consuming: %f W\n", iVport, Iload, Pload);

printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
printf("Performing a 10 second trace capture\n");
printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
```

```
	// Capture a 10 sec trace

	iRtn = pda300_StartTraceCapture(pszTraceFilePath);
	checkRtn(iRtn);

	time(&startTime);
	time(&currentTime);

	// loop for 10 seconds
	while ((currentTime - startTime) <= 10)
	{
		time(&currentTime);
	}

	iRtn = pda300_StopTraceCapture();
	checkRtn(iRtn);

	printf ("Trace file stored in %s\n", pszTraceFilePath);

	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
	printf("Running the built-in 802.3at test - one quadrant\n");
	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

	// These are examples of running the built-in 802.3at test.

	// This mode measures all parameters for Alt-A, MDI, and
	// unpowered parameters for the other three quadrants,
	// Alt-A MDI-X, Alt-B MDI, and Alt-B MDI-X
	iRtn = pda300_Run802dot3ATtest(ONE_QUADRANT, pszOneQuadFilePath);
	checkRtn(iRtn);

	if (iRtn == PDA300_NO_ERROR)
		printf ("One quadrant 802.3at test results stored in %s\n", pszOneQuadFilePath);

	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
	printf("Running the built-in 802.3at test - four quadrants\n");
	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

	// This mode measures all parameters for all quadrants.
	iRtn = pda300_Run802dot3ATtest(FOUR_QUADRANTS, pszFourQuadFilePath);
	checkRtn(iRtn);

	if (iRtn == PDA300_NO_ERROR)
		printf ("Four quadrant 802.3at test results stored in %s\n", pszFourQuadFilePath);

	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
	printf("Done!\n");
	printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");

	// Close the connection to the instrument, re-enabling the front panel
	iRtn = pda300_Disconnect();
	checkRtn(iRtn);

	return 0;
}

void checkRtn (int iRtn)
{
	int iDisplayErrorMsg = 1;
	char *pszMsg;
	char szErrMsg[LENGTH_PDA300_ERROR_BUF];

	switch (iRtn)
```

```
	{
	case PDA300_NO_ERROR:
		pszMsg = "iRtn: PDA300_NO_ERROR";
		iDisplayErrorMsg = 0;
		break;
	case PDA300_SERIAL_COMM_ERROR:
		pszMsg = "iRtn: PDA300_SERIAL_COMM_ERROR";
		break;
	case PDA300_INVALID_FW_VERSION:
		pszMsg = "iRtn: PDA300_INVALID_FW_VERSION";
		break;
	case PDA300_INVALID_PARAMETER_VALUE:
		pszMsg = "iRtn: PDA300_INVALID_PARAMETER_VALUE";
		break;
	case PDA300_INVALID_ICLASS:
		pszMsg = "iRtn: PDA300_INVALID_ICLASS";
		break;
	case PDA300_NULL_POINTER:
		pszMsg = "iRtn: PDA300_NULL_POINTER";
		break;
	case PDA300_FILE_ERROR:
		pszMsg = "iRtn: PDA300_FILE_ERROR";
		break;
	case PDA300_MENU_ERROR:
		pszMsg = "iRtn: PDA300_MENU_ERROR";
		break;
	case PDA300_NOT_IN_LM_MENU:
		pszMsg = "iRtn: PDA300_NOT_IN_LM_MENU";
		break;
	case PDA300_FAST_SAMPLE_RUNNING:
		pszMsg = "iRtn: PDA300_FAST_SAMPLE_RUNNING";
		break;
	case PDA300_FAST_SAMPLE_NOT_RUNNING:
		pszMsg = "iRtn: PDA300_FAST_SAMPLE_NOT_RUNNING";
		break;
	case PDA300_TRACE_ERROR:
		pszMsg = "iRtn: PDA300_TRACE_ERROR";
		break;
	case PDA300_VPORT_NOT_ENABLED:
		pszMsg = "iRtn: PDA300_VPORT_NOT_ENABLED";
		break;
	case PDA300_TRACE_FILE_FORMAT_ERROR:
		pszMsg = "iRtn: PDA300_TRACE_FILE_FORMAT_ERROR";
		break;
	case PDA300_TEST_NEVER_COMPLETED:
		pszMsg = "iRtn: PDA300_TEST_NEVER_COMPLETED";
		break;
	default:
		printf("DEFAULT CASE: iRtn value: %d\n", iRtn);
		pszMsg = "iRtn: UNRECOGNIZED";
		break;
	}

	if (iDisplayErrorMsg == 1)
	{
		pda300_GetErrMessage(szErrMsg, LENGTH_PDA300_ERROR_BUF);
		printf("%s\n%s\n", pszMsg, szErrMsg);
	}
}
```

4.2. Visual Basic 6 Code Example

```
Private Sub TestAPI()

    Dim iRtn As Long
    Dim eAltSetting As pda300AltSetting
    Dim strTest As String
    Dim strBuffer As String * LENGTH_PDA300_ERROR_BUF

    iRtn = pda300_SetAlt(pda300AltSetting.ALT_B)
    iRtn = pda300_GetAlt(eAltSetting)

    iRtn = pda300_GetErrMessage(strBuffer, LENGTH_PDA300_ERROR_BUF)

    strTest = TrimAPIString(strBuffer)

End Sub
```

4.3. Visual Basic .NET Code Example

NOTE: you must include the provided source module pda300API.vb in your project. The file pda300API.vb contains the Visual Basic .NET compatible declarations for the API function calls.

```
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'
' pda300API_example.vb : illustrates how to use the PDA-300 API Library
' with VB.NET Console Application
'
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'
' Console output from the example program
'
'ALT is set to ALT-B
'Polarity is set to MDI-X
'Classification will use TWO Events
'Vport level is set to 54 V
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Performing unpowered measurements
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Rdet: 24.049999 k is in range
'Cdet: 0.106000 uF is in range
'PD Class is: 4  measured Iclass: 40.000000 mA
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Applying Power
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Vport is ON
'With Vport 54 V, PD is drawing: 19.600000 mA and consuming: 1.050000 W
'Reducing Vport
'With Vport 46 V, PD is drawing: 21.700001 mA and consuming: 0.990000 W
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Removing Power, changing Alt and Polarity
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Performing unpowered measurements
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Rdet: 23.940001 k is in range
'Cdet: 0.112000 uF is in range
'PD Class is: 4  measured Iclass: 40.099998 mA
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Applying Power
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Vport is ON
'With Vport 46 V, PD is drawing: 21.700001 mA and consuming: 0.990000 W
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Performing a 10 second trace capture
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Trace file stored in /temp/pda300_trace.csv
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Running the built-in 802.3at test - one quadrant
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'One quadrant 802.3at test results stored in /temp/oneQuadrantResults.csv
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Running the built-in 802.3at test - four quadrants
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Four quadrant 802.3at test results stored in /temp/fourQuadrantResults.csv
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'Done!
'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

'%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Module Module1

    Sub Main()
        Dim iRtn As Integer
        Dim eAlt As pda300AltSetting
        Dim eMDI As pda300MdiSetting
        Dim eEvents As pda300ClassEvents

        Dim iVport As Integer
        iVport = 0
        Dim eVportState As pda300VportState
        Dim COMPortName As System.Text.StringBuilder = New System.Text.StringBuilder(10)
        ' NOTE: change the COM port to the one that the instrument is connected to
        '       on your system.
        COMPortName.Append("COM1")
        Dim Rdet As Single
        Dim Cdet As Single
        Dim Iclass As Single
        Dim pdClass As Integer
        Dim Iload As Single
        Dim Pload As Single
        Dim Delay As Integer

        Dim pszDbgFilePath As System.Text.StringBuilder = New System.Text.StringBuilder(20)
        Dim pszTraceFilePath As System.Text.StringBuilder = New System.Text.StringBuilder(20)
        Dim pszOneQuadFilePath As System.Text.StringBuilder = New System.Text.StringBuilder(20)
        Dim pszFourQuadFilePath As System.Text.StringBuilder = New System.Text.StringBuilder(20)
        ' NOTE: change these paths to a legal path on your system.
        pszDbgFilePath.Append("debug_output.txt")
        pszTraceFilePath.Append("pda300_trace.csv")
        pszOneQuadFilePath.Append("oneQuadrantResults.csv")
        pszFourQuadFilePath.Append("fourQuadrantResults.csv")

        'Enable lines below to create a Debug file
        'iRtn = pda300_SetDebugPath(pszDbgFilePath)
        'checkRtn(iRtn)

        ' Open a connection to the instrument, locking out the front panel
        iRtn = pda300_Connect(COMPortName)
        checkRtn(iRtn)

        ' Define the ALT setting
        iRtn = pda300_SetAlt(pda300AltSetting.ALT_B)
        checkRtn(iRtn)

        'Interrogate the library to check the ALT setting
        iRtn = pda300_GetAlt(eAlt)
        checkRtn(iRtn)

        If (eAlt = pda300AltSetting.ALT_A) Then
            Console.WriteLine("ALT is set to ALT-A" + vbCrLf)
        Else
            Console.WriteLine("ALT is set to ALT-B" + vbCrLf)
        End If
        ' Define the polarity setting
        iRtn = pda300_SetMdi(pda300MdiSetting.MDIX)
        checkRtn(iRtn)

        ' Interrogate the library to check the polarity setting
        iRtn = pda300_GetMdi(eMDI)
        checkRtn(iRtn)

        If (eMDI = pda300MdiSetting.MDI) Then
            Console.WriteLine("Polarity is set to MDI" + vbCrLf)
```

```
    Else
       Console.WriteLine("Polarity is set to MDI-X" + vbCrLf)
    End If
    ' Define the number of event to use during Classification
    iRtn = pda300_SetEvents(pda300ClassEvents.TWO_EVENTS)
    checkRtn(iRtn)

    ' Interrogate the library to check the number of events setting
    iRtn = pda300_GetEvents(eEvents)
    checkRtn(iRtn)

    If (eEvents = pda300ClassEvents.ONE_EVENT) Then
       Console.WriteLine("Classification will use ONE Event" + vbCrLf)
    Else
       Console.WriteLine("Classification will use TWO Events" + vbCrLf)
    End If

    ' Define the Vport voltage level when power is applied
    iRtn = pda300_SetVportLevel(54)
    checkRtn(iRtn)

    ' Interrogate the library to check the voltage level setting
    iRtn = pda300_GetVportLevel(iVport)
    checkRtn(iRtn)

    Console.WriteLine("Vport level is set to " + iVport.ToString + "V" + vbCrLf)

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Performing unpowered measurements" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' Measure Rdet
    iRtn = pda300_MeasRdet(Rdet)
    checkRtn(iRtn)

    If (Rdet < 23.2 Or Rdet > 24.8) Then
       Console.WriteLine("Rdet: " + Rdet.ToString + "k is out of range" + vbCrLf)
    Else
       Console.WriteLine("Rdet: " + Rdet.ToString + "k is in range" + vbCrLf)
    End If
    ' Measure Cdet
    iRtn = pda300_MeasCdet(Cdet)
    checkRtn(iRtn)

    If (Cdet < 0.09 Or Cdet > 0.15) Then
       Console.WriteLine("Cdet: " + Cdet.ToString + "uF is out of range" + vbCrLf)
    Else
       Console.WriteLine("Cdet: " + Cdet.ToString + "uF is in range" + vbCrLf)
    End If
    ' Determine what Class the PD represents itself as
    iRtn = pda300_MeasClass(Iclass, pdClass)
    checkRtn(iRtn)

    If (iRtn = PDA300_INVALID_ICLASS) Then
       Console.WriteLine("Invalid Iclass: " + Iclass.ToString + "mA cannot determine PD Class" + vbCrLf)
    Else
       Console.WriteLine("PD Class is: " + pdClass.ToString + ",  measured Iclass: " + Iclass.ToString + "mA" +
vbCrLf)
    End If

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Applying Power" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
```

```
      ' Apply power
      iRtn = pda300_SetVportState(pda300VportState.VPORT_ON)
      checkRtn(iRtn)

      ' Interrogate the library to check the state of Vport
      iRtn = pda300_GetVportState(eVportState)
      checkRtn(iRtn)

      If (eVportState = pda300VportState.VPORT_ON) Then
         Console.WriteLine("Vport is ON" + vbCrLf)
      Else
         Console.WriteLine("Vport is OFF" + vbCrLf)
      End If

      ' Measure the current demand and power consumed by the PD
      iRtn = pda300_MeasPowerDraw(Iload, Pload)
      checkRtn(iRtn)

      Console.WriteLine("With Vport " + iVport.ToString + "V, PD is drawing: " + Iload.ToString + "mA  and
consuming: " + Pload.ToString + "W" + vbCrLf)

      Console.WriteLine("Reducing Vport" + vbCrLf)

      ' Reduce Vport to 46V
      iRtn = pda300_AdjustVportLevel(46)
      checkRtn(iRtn)

      ' Interrogate the library to check the voltage level setting
      iRtn = pda300_GetVportLevel(iVport)
      checkRtn(iRtn)

      ' Measure the current demand and power consumed by the PD
      iRtn = pda300_MeasPowerDraw(Iload, Pload)
      checkRtn(iRtn)

      Console.WriteLine("With Vport " + iVport.ToString + "V, PD is drawing: " + Iload.ToString + "mA and
consuming: " + Pload.ToString + "W" + vbCrLf)

      Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
      Console.WriteLine("Removing Power, changing Alt and Polarity" + vbcrlf)
      Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

      ' Remove power
      iRtn = pda300_SetVportState(pda300VportState.VPORT_OFF)
      checkRtn(iRtn)

      ' Change settings
      iRtn = pda300_SetAlt(pda300AltSetting.ALT_A)
      checkRtn(iRtn)

      iRtn = pda300_SetMdi(pda300MdiSetting.MDI)
      checkRtn(iRtn)

      Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
      Console.WriteLine("Performing unpowered measurements" + vbcrlf)
      Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

      ' Measure Rdet
      iRtn = pda300_MeasRdet(Rdet)
      checkRtn(iRtn)

      If (Rdet < 23.2 Or Rdet > 24.8) Then
         Console.WriteLine("Rdet: " + Rdet.ToString + "k is out of range" + vbCrLf)
      Else
```

```
        Console.WriteLine("Rdet: " + Rdet.ToString + "k is in range" + vbCrLf)
    End If
    ' Measure Cdet
    iRtn = pda300_MeasCdet(Cdet)
    checkRtn(iRtn)

    If (Cdet < 0.09 Or Cdet > 0.15) Then
        Console.WriteLine("Cdet: " + Cdet.ToString + "uF is out of range" + vbCrLf)
    Else
        Console.WriteLine("Cdet: " + Cdet.ToString + "uF is in range" + vbCrLf)
    End If
    ' Determine what Class the PD represents itself as
    iRtn = pda300_MeasClass(Iclass, pdClass)
    checkRtn(iRtn)

    If (iRtn = PDA300_INVALID_ICLASS) Then
        Console.WriteLine("Invalid Iclass: " + Iclass.ToString + "mA cannot determine PD Class" + vbCrLf)
    Else
        Console.WriteLine("PD Class is: " + pdClass.ToString + ", measured Iclass: " + Iclass.ToString + "mA" +
vbCrLf)
    End If

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Applying Power" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' Apply power...NOTE that the Vport level will be the one that was
    ' last defined.
    iRtn = pda300_SetVportState(pda300VportState.VPORT_ON)
    checkRtn(iRtn)

    ' Interrogate the library to check the state of Vport
    iRtn = pda300_GetVportState(eVportState)
    checkRtn(iRtn)

    If (eVportState = pda300VportState.VPORT_ON) Then
        Console.WriteLine("Vport is ON" + vbCrLf)
    Else
        Console.WriteLine("Vport is OFF" + vbCrLf)
    End If
    ' Interrogate the library to check the voltage level setting
    iRtn = pda300_GetVportLevel(iVport)
    checkRtn(iRtn)

    ' Measure the current demand and power consumed by the PD
    iRtn = pda300_MeasPowerDraw(Iload, Pload)
    checkRtn(iRtn)

    Console.WriteLine("With Vport " + iVport.ToString + "V," + " PD is drawing: " + Iload.ToString + "mA" + " and
consuming: " + Pload.ToString + "W" + vbCrLf)

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Performing a 10 second trace capture" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' Capture a 10 sec trace

    iRtn = pda300_StartTraceCapture(pszTraceFilePath)
    checkRtn(iRtn)

    ' loop for 10 seconds
    Delay = 1
    While Delay <= 10
        Delay = Delay + 1
```

```
        System.Threading.Thread.Sleep(1000) 'Wait 1 second
    End While


    iRtn = pda300_StopTraceCapture()
    checkRtn(iRtn)

    Console.WriteLine("Trace file stored in " + pszTraceFilePath.ToString() + vbCrLf)

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Running the built-in 802.3at test - one quadrant" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' These are examples of running the built-in 802.3at test.

    ' This mode measures all parameters for Alt-A, MDI, and
    ' unpowered parameters for the other three quadrants,
    ' Alt-A MDI-X, Alt-B MDI, and Alt-B MDI-X
    iRtn = pda300_Run802dot3ATtest(pda300Quadrants.ONE_QUADRANT, pszOneQuadFilePath)
    checkRtn(iRtn)

    If (iRtn = PDA300_NO_ERROR) Then
        Console.WriteLine("One quadrant 802.3at test results stored in " + pszOneQuadFilePath.ToString() + vbCrLf)
    End If

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Running the built-in 802.3at test - four quadrants" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' This mode measures all parameters for all quadrants.
    iRtn = pda300_Run802dot3ATtest(pda300Quadrants.FOUR_QUADRANTS, pszFourQuadFilePath)
    checkRtn(iRtn)

    If (iRtn = PDA300_NO_ERROR) Then
        Console.WriteLine("Four quadrant 802.3at test results stored in " + pszFourQuadFilePath.ToString() + vbCrLf)
    End If

    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)
    Console.WriteLine("Done!" + vbcrlf)
    Console.WriteLine("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%" + vbcrlf)

    ' Close the connection to the instrument, re-enabling the front panel
    iRtn = pda300_Disconnect()
    checkRtn(iRtn)

    Console.WriteLine("Exiting Program")
    System.Threading.Thread.Sleep(5000) 'Wait 5 seconds


End Sub


Sub checkRtn(ByVal iRtn As Integer)

    Dim pszMsg As String
    Dim szErrMsg As System.Text.StringBuilder = New
System.Text.StringBuilder(LENGTH_PDA300_ERROR_BUF)

    If iRtn = PDA300_NO_ERROR Then
        pszMsg = ""
        'pszMsg = "iRtn: PDA300_NO_ERROR"   'Comment in to report Non-Errors
    ElseIf iRtn = PDA300_SERIAL_COMM_ERROR Then
        pszMsg = "iRtn: PDA300_SERIAL_COMM_ERROR"
    ElseIf iRtn = PDA300_INVALID_FW_VERSION Then
```

```
            pszMsg = "iRtn: PDA300_INVALID_FW_VERSION"
        ElseIf iRtn = PDA300_INVALID_PARAMETER_VALUE Then
            pszMsg = "iRtn: PDA300_INVALID_PARAMETER_VALUE"
        ElseIf iRtn = PDA300_INVALID_ICLASS Then
            pszMsg = "iRtn: PDA300_INVALID_ICLASS"
        ElseIf iRtn = PDA300_NULL_POINTER Then
            pszMsg = "iRtn: PDA300_NULL_POINTER"
        ElseIf iRtn = PDA300_FILE_ERROR Then
            pszMsg = "iRtn: PDA300_FILE_ERROR"
        ElseIf iRtn = PDA300_MENU_ERROR Then
            pszMsg = "iRtn: PDA300_MENU_ERROR"
        ElseIf iRtn = PDA300_NOT_IN_LM_MENU Then
            pszMsg = "iRtn: PDA300_NOT_IN_LM_MENU"
        ElseIf iRtn = PDA300_FAST_SAMPLE_RUNNING Then
            pszMsg = "iRtn: PDA300_FAST_SAMPLE_RUNNING"
        ElseIf iRtn = PDA300_FAST_SAMPLE_NOT_RUNNING Then
            pszMsg = "iRtn: PDA300_FAST_SAMPLE_NOT_RUNNING"
        ElseIf iRtn = PDA300_TRACE_ERROR Then
            pszMsg = "iRtn: PDA300_TRACE_ERROR"
        ElseIf iRtn = PDA300_VPORT_NOT_ENABLED Then
            pszMsg = "iRtn: PDA300_VPORT_NOT_ENABLED"
        ElseIf iRtn = PDA300_TRACE_FILE_FORMAT_ERROR Then
            pszMsg = "iRtn: PDA300_TRACE_FILE_FORMAT_ERROR"
        ElseIf iRtn = PDA300_TEST_NEVER_COMPLETED Then
            pszMsg = "iRtn: PDA300_TEST_NEVER_COMPLETED"
        Else
            Console.WriteLine("DEFAULT CASE: iRtn value: " + iRtn.ToString + vbCrLf)
            pszMsg = "iRtn: UNRECOGNIZED"
        End If

        pda300_GetErrMessage(szErrMsg, LENGTH_PDA300_ERROR_BUF)
        If pszMsg <> "" Then
            Console.WriteLine(pszMsg + vbCrLf) '(Actual message test -->) + szErrMsg.ToString() + vbCrLf)
        End If
    End Sub

End Module
```

4.4. LabView Code Example

LabView example programs illustrating the use of the PDA-300 API VIs are located under the paths on the **Sifos PDA-300** CD-ROM:

```
\API_Files\LabView\Error_In_Error_Out_form\pda300_example.vi
\API_Files\LabView\Original_Flat_Sequence_form\pda300_example.vi
```

NOTE: the examples were created with LabView 8.20.

NOTE: this author is not a skilled LabView user, and the examples were used primarily as a test harness to verify that the API VIs are defined correctly. The example program does not illustrate a complete test application – for example, it does not include logic to test each of the VIs error return, and modify application flow if an error occurs. It is expected that the user will use the VIs with the necessary flow control logic, as appropriate for their application.