Programming Guide for the

PULSE PROFILE APPLICATION

MODELS LB480A/LB680A

TABLE OF CONTENTS

NOTICES

SAFETY

A ***WARNING*** indicates a potential hazard that could completely damage the product. Do not continue until you fully understand the meaning.

A ***CAUTION*** indicates a potential hazard that could partially damage the product. Do not continue until you fully understand the meaning.

A ***NOTE*** provides additional, pertinent information related to the operation of the product.

CONFORMITY

WEEE Compliant
RoHS Compliant
USB 2.0 Compliant

DISCLAIMER

The information contained in this document is subject to change without notice. There is no guarantee as to the accuracy of the material presented or its application. Any errors of commission or omission will be corrected in subsequent revisions or made available by errata.

WARRANTY

See the warranty section of the Product Manual for details.

DOCUMENT NUMBER

Not Assigned (Reference Programming Guide for the Pulse Profile Application Models LB480A/LB680A).

CONTACT INFORMATION

LadyBug Technologies LLC
3317 Chanate Road
Suite 2F
Santa Rosa, CA 95404
Phone 707.546.1050
Fax 707.237.6724
www.ladybug-tech.com

Introduction

This document is a programming guide for the LB480A and LB680A power sensors using the ***Pulse Profile Application***. Note that all sensor models can run the ***Power Meter Application***. However, at this time, the LB480A and LB680A are capable of making pulse profile measurements. There are functions listed specific to the ***Pulse Profiling Application*** using the LB480A and LB680A. There are also some factory calls listed for general information purposes.

This document applies primarily to pulse profiling. All of the calls detailed that apply exclusively to pulse profiling are prepended with a *"PP_".* For example, the call to set the sweep time is *PP_SetSweepTime* just as the call to get the sweep time is *PP_GetSweepTime*. Clearly, setting the sweep time has no value for a power meter application. If you have examined the companion programming manual for the power meter application, you will see that the function calls in that manual are all prepended with *"LB_".*

While *PP_* calls are intended for pulse profiling, *LB_* calls include both general purpose calls and power meter calls. The general purpose calls apply to both pulse profile and power meter applications. The general purpose calls that are of use in pulse profiling applications are listed at the beginning of the reference portion of this document. This list includes such calls as *LB_Initialize_Addr* (used to initialize a sensor); and *LB_BlinkLED_Addr* (used to identify a sensor physically).

The programmatic interface consists of a dynamic link library or DLL. The name of the DLL is *LB_API2.DLL*. This library uses the WinAPI or "_stdcall" calling convention. We have chosen a DLL and this calling convention because they provide greater access to more of the most common environments. This DLL is located in the Ladybug application directory. The name of the default application directory is *"C:\Program Files\Ladybug Technologies LLC\Ladybug Pulse Profiling Application".*

Included in the product installation are various demonstration programs. The programs are written in VB 6.0 and VB.NET and C#. Almost all functions are demonstrated in these applications. The name of the applications generally have the term "TestHarness" embedded in the name. A C# test harness that demonstrates the pulse profiling calls is named "PulseProfiling_TestHarness_Csh". This test harness includes functions for all of the power meter functions plus the pulse profiling functions including retrieving and displaying a trace, triggering, offsets, etc. Other sample code is also available.

Unfortunately, variable type LONG (or long) has changed with Visual Studio .NET. A "long" in most .NET languages is 64 bits long. However, the "long" in these prototypes are 32 bits long.

Making a Simple Measurement

The purpose of this section is to get you up and running quickly. We will cover the simplest case of making a CW measurement using VB 6.0, VB.NET and C SHARP.

NOTE: Before starting, install the application provided on the product media. Then connect one sensor to the PC as instructed in the Quick Start Guide. Make sure the system is functional by making a few basic measurements using the GUI.

The following VB.6, VB.NET and C SHARP code makes a simple CW measurement. The VB.NET and C SHARP were created using Microsoft Visual Studio 2005. This code assumes that a single sensor has been connected to your computer and has proven functional. If you are using an earlier version of Visual Studio.NET, the VB.NET and C SHARP code may need some tweaking as a direct copy and paste may not work. In any event, the changes should be minor.

Writing the Code:

Start the code by creating a default Windows application. Place three buttons and one label on the window or form. Name the buttons as shown below:

- cmdGetAddress
- cmdInitialize
- cmdMeasure

Name the label “lblCW”. Copy the appropriate set of code (or portions if you prefer) from the pages below.

Explanation of the Code:

In each case (VB 6, VB.Net and C SHARP) the same approach has been taken. First, the address of the instrument is obtained when *cmdGetAddress* is clicked. We use the call “LB_GetAddress_Idx”. The name of this call can be interpreted as “get the address using the index.” We are using the first sensor in this case, or the sensor with an index of 1.

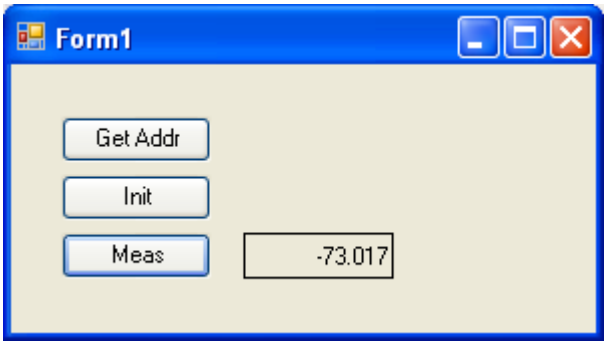
We can initialize the sensor using the address from the first call. This is accomplished by clicking the second button on the form. This makes the call “LB_InitializeSensor_Addr”. This call can be interpreted as “initialize the sensor using the address”. Initialization causes the calibration constants and other information for the sensor to be transferred to the PC. Now that we have the address and we have initialized the sensor we can make a measurement.

A CW measurement is made by using "LB_CWMeasure”. This is done when the third button is clicked. The result of the measurement is converted to text and placed in the label. This call requires the address acquired in the first button click. It also requires that the sensor be initialized as done in the second button click.

In this API most calls are designed for use with the address. Once we have the address and we have initialized the sensor we can remeasure as often as we like. We can also change state and remeasure.

Using the Application:

To use the application you just coded, compile it and run. The window should look similar to the one below:



Then follow the sequence outlined below:

- Click the “Get Addr” or `cmdGetAddress` button
- Click the “Init” or `cmdinitize` button - wait for the message indicating initialization is complete. This typically takes about 5 seconds.
- Click the “Meas” or `cmdMeasure` button (click this button as often as you like). A measurement should appear in the label. Now that the instrument has been initialized the button can be clicked repeatedly.

A few items that may be of interest to some programmers are:

- “Long” in VB 6.0 is equivalent to an “Integer” in VB.NET and “int” in C SHARP.
- The default ByRef/ByVal are switched when going from VB 6 to VB.NET and C SHARP. We have taken the approach of explicitly including the ByRef/ByVal declarations in all code. We highly recommend this practice.
- Structures in VB 6.0 allowed the embedding of fixed arrays. This is (was) commonly used for transferring complex data types. The exact capability has not been duplicated in VB.NET and C SHARP. While VB.NET does have the following type of declaration that can be used inside a structure:

  <VBFixedArray(6)> Dim SerialNumber() As Byte

  It seems able to be passed via a _stdcall for simple structures only. It does not work for more complex structures in our experience.

NOTE: If you are using an earlier version of Visual Studio.NET you may need to modify the code to some extent.

VB 6.0 Code

```
Option Explicit

Private Declare Function LB_SensorCnt Lib _
                           "LB_API2.dll" () _
                           As Long

Private Declare Function LB_GetAddress_Idx _
                           Lib "LB_API2.dll" ( _
                           ByVal addr As Long) _
                           As Long

Private Declare Function LB_InitializeSensor_Addr _
                           Lib "LB_API2.dll" ( _
                           ByVal addr As Long) _
                           As Long

Private Declare Function LB_MeasureCW _
                           Lib "LB_API2.dll" ( _
                           ByVal addr As Long, _
                           ByRef CW As Double) As Long

Dim m_Addr As Long

Private Sub cmdGetAddress_Click()
    If LB_SensorCnt() > 0 Then
        m_Addr = LB_GetAddress_Idx(1)
    End If
End Sub

Private Sub cmdInitialize_Click()
    If LB_InitializeSensor_Addr(m_Addr) > 0 Then
        MsgBox ("Initialization OK")
    End If
End Sub

Private Sub cmdMeasure_Click()
    Dim CW As Double, rslt As Long

    rslt = LB_MeasureCW(m_Addr, CW)
    If rslt > 0 Then lblCW.Caption = Format(CW, "###0.0###")
End Sub
```

VB.NET Code (Visual Studio 2005)

```
Public Class Form1

    Public Declare Function LB_SensorCnt Lib _
                                "LB_API2.dll" () _
                                As Integer

    Public Declare Function LB_GetAddress_Idx _
                                Lib "LB_API2.dll" ( _
                                ByVal addr As Integer) _
                                As Integer

    Public Declare Function LB_InitializeSensor_Addr _
                                Lib "LB_API2.dll" ( _
                                ByVal addr As Integer) _
                                As Integer

    Public Declare Function LB_MeasureCW _
                                Lib "LB_API2.dll" ( _
                                ByVal addr As Integer, _
                                ByRef CW As Double) As Integer

    Dim m_Addr As Integer

    Private Sub cmdGetAddress_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles cmdGetAddress.Click
        If LB_SensorCnt() > 0 Then
            m_Addr = LB_GetAddress_Idx(1)
        End If
    End Sub

    Private Sub cmdInitialize_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles cmdInitialize.Click
        If LB_InitializeSensor_Addr(m_Addr) > 0 Then
            MsgBox("Initialization OK")
        End If
    End Sub

    Private Sub cmdMeasure_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles cmdMeasure.Click

        Dim CW As Double, rslt As Long

        rslt = LB_MeasureCW(m_Addr, CW)
        If rslt > 0 Then lblCW.Text = Format(CW, "###0.0###")
    End Sub
End Class
```

C SHARP Code (Visual Studio 2005)

```
using Microsoft.VisualBasic;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Drawing;
using System.Diagnostics;
using System.Windows.Forms;
namespace SimpleMeasurement
{
    public partial class Form1
    {
        public Form1()
        {
            InitializeComponent();
            cmdGetAddress.Click += new System.EventHandler( cmdGetAddress_Click );
            cmdInitialize.Click += new System.EventHandler( cmdInitialize_Click );
            cmdMeasure.Click += new System.EventHandler( cmdMeasure_Click );
        }

        [System.Runtime.InteropServices.DllImport("LB_API2.dll")]
        public static extern int LB_SensorCnt();

        [System.Runtime.InteropServices.DllImport("LB_API2.dll")]
        public static extern int LB_GetAddress_Idx( int addr );

        [System.Runtime.InteropServices.DllImport("LB_API2.dll")]
        public static extern int LB_InitializeSensor_Addr( int addr );

        [System.Runtime.InteropServices.DllImport("LB_API2.dll")]
        public static extern int LB_MeasureCW( int addr, ref double CW );

        public int m_Addr;

        private void cmdGetAddress_Click( System.Object sender, System.EventArgs e )
        {
            if ( LB_SensorCnt() > 0 )
            {
                m_Addr = LB_GetAddress_Idx( 1 );
            }
        }

        private void cmdInitialize_Click( System.Object sender, System.EventArgs e )
        {
            if ( LB_InitializeSensor_Addr( m_Addr ) > 0 )
            {
                Interaction.MsgBox( "Initialization OK",
(Microsoft.VisualBasic.MsgBoxStyle)(0), null );
            }
        }
```

```
        private void cmdMeasure_Click( System.Object sender, System.EventArgs e )
        {

            double CW = 0; long rslt = 0;

            rslt = LB_MeasureCW( m_Addr, ref CW );
            if ( rslt > 0 )
            {
                lblCW.Text = Strings.Format( CW, "###0.0###" );
            }
        }
    }
}
```

Addressing and Communicating with Sensors

In the past, communicating with instrumentation via GPIB was accomplished by using addresses. This approach provided a great advantage to those writing test code. In particular, it allowed the software to be written in a way that was more flexible. GPIB addresses were typically set at the front panel of the instrument or using switches on the back of the instrument (and sometimes inside the instrument).

An inspection of a Ladybug power sensor brings up and important point. Ladybug sensors do not have switches or front panels. So, how do you control or communicate with a Ladybug sensor? The following questions become important:

- How do I discover the address of my sensor?
- How do I set or change the address of a sensor?
- How do I know which sensor is at which address if I have several sensors connected in a system?
- What do I do about address conflicts?
- Is there a means of identifying a particular sensor?
- How do I deal with this in my code?

NOTE: We have a number of applications available on the product CD to set and check instrument addresses. The applications and the code for these applications are on the CD. The code is available to aid the development of applications. Feel free to examine these applications to help reinforce this explanation.

The first step in communicating with an instrument is to identify it uniquely. The best way to do this is to look at the physical identification present on the sensor. If you look at the back of the sensor you will see a serial number. You will also note that there is a green LED (power light). We provide function calls to support the following:

- Allows collection of all sensor identification information (index, serial number and address)
- Allows the address to be obtained by serial number or index
- Allows the address to be set/changed using the index, serial number or current address
- Allows the serial number to be retrieved using the index or address
- Allows the index to be retrieved using the serial number or address
- Allows you to blink the LED on a specific sensor
- Allows you to determine if an address conflict exists
- Allows you to determine if changing an address will cause an address conflict

The following is a discussion of communicating with Ladybug sensors. Hopefully, the obvious questions will be answered first and by doing so you will be able to get on with your own work. As noted before, we have provided an application for this purpose and the code is available on the product CD.

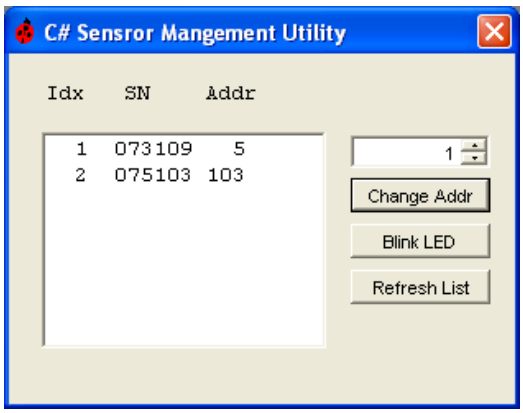
We will break this in to the following two steps:

1. Setting the address and identifying the sensor(s).
2. Communicating with the sensor(s).

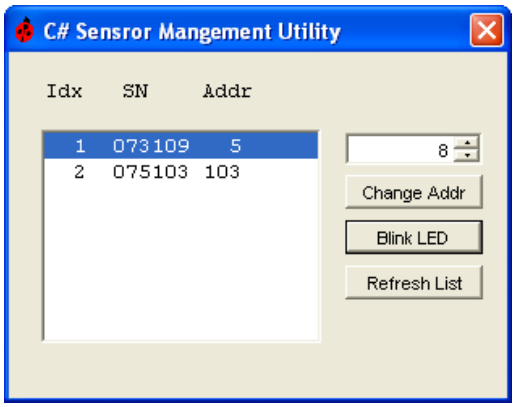
Step 1 - Setting the Address(es)

Open the “Managing Addresses” application provided to accomplish the first step. This application should be visible in the Ladybug menu (*Start* > *Ladybug* > *Addresses*). You should see the window below when the application starts.

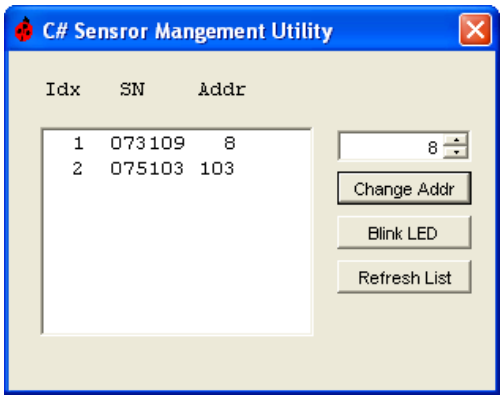
You should see a list of sensor(s) currently attached to your computer. Each sensor is represented by an index; a serial number (stamped on the back of the sensor); and an address.



Select a sensor as shown below. Use the up/down arrows to set the desired address. We have chosen to change the address of the sensor with a serial number of “073109” in the picture below. The picture indicates that the address will be changed from 5 to 8. Use the “Blink LED” button to ensure you are addressing the correct sensor.



Click “Change Addr” to change the address. The address of the sensor will be updated as shown below and then the list will be updated.



Select the sensor in the list box and click “Blink LED” to identify the sensor whose address was just changed. The sensor’s LED should blink four times in quick succession.

Close the application once you have the sensor set to the address of choice. *The address is set in non-volatile memory so losing power after the address is set or moving the sensor from system to system is not an issue*. To change the address, connect the sensor to the system and re-run the application.

NOTE: All this can be accomplished programmatically in your code with just a few calls. The code for this application is in the examples directory on the media in VB 6.0, VB.NET and C SHARP.

Step 2 - Communicating with Your Sensor(s)

As you look at the API provided (see the declarations in the sample VB 6.0, VB.NET and C SHARP projects), you will note that making measurements and setting various parameters requires the address. Some of the management calls use the serial number or index, but most of the API calls use the address exclusively.

You know how to communicate with the sensor using the index and address if you followed “Making a Simple Measurement” in the previous section. Review the next section entitled “More Detail” if this has not met your needs. The address was requested first by using the index in the previous example. You can skip this step since you already set the address. Your code can initialize the instrument using the address you just setup - then make a measurement!

More Detail

Sensors can be identified three ways: The first is temporary (the index) and determined by the system driver when the device is connected. The second is permanent and determined by the factory (the serial number). The third method of identification is the address. You have complete control over the address and you can assign any legitimate address (1-255) to any sensor.

The address is stored in non-volatile memory so it is not lost when the sensor is disconnected or your system is powered down. Note that address conflicts may arise during the process of reassigning sensor addresses. Some functions do not require the index, address or serial number. They are listed below:

- LB_SensorCnt - returns the number of sensors connected to the system
- LB_SensorList - returns a list of sensors (index, serial number and address)
- LB_AddressConflictExists - returns a 1 if an address conflict exists, returns a 0 otherwise

The index is an arbitrary number that is assigned by order of identification. The index of the first sensor detected by the system is 1. The index of the second sensor is 2 and so on. Typically, you will find the index less useful than address and serial number although it is provided for completeness sake. The index is most useful when coupled with LB_SensorCnt. The index of the sensors will be between 1 and the sensor count assuming the sensor count is greater than zero.

For instance, if the sensor count is three, the first sensor discovered will have an index of 1; the second sensor will have an index of 2; and the third sensor will have an index of 3. You can get or set the address and retrieve the serial number using the index and you can cause the LED to blink based on the index.

The functions applicable to index are listed below:

- LB_GetAddress_Idx - returns the address of the unit
- LB_SetAddress_Idx - sets the address of the unit
- LB_GetModelNumber_Idx - get a number indicating the model number (1-3)
- LB_GetSerNo_Idx - returns the serial number of the unit
- LB_InitializeSensor_Idx - initializes the sensor (causes calibration data to be downloaded)
- LB_BlinkLED_Idx - blinks the LED (useful in identifying the units physically)

The serial number is immutable and set at the factory. You can get the address or index using the serial number. You can also change the address and cause the LED to blink. In addition, the serial number is required to get option information and to change the calibration due date.

The functions applicable to serial number are listed below:

- LB_GetAddress_SN – returns the address of the unit with the serial number
- LB_SetAddress_SN – sets the address of the unit with the serial number
- LB_GetModelNumber_SN – gets the model number (1-3)
- LB_IsSensorConnected_SN – indicates if a unit with the serial number is attached
- LB_GetIndex_SN – gets the index of the unit with the serial number
- LB_InitializeSensor_SN – initializes the sensor (causes calibration data to be downloaded)
- LB_BlinkLED_SN – blinks the LED (useful in identifying the units physically)
- LB_SetCalDueDate – sets the cal due date of the unit (stored in non-volatile memory)
- LB_GetCalDueDate – gets the cal due date

Finally, we can discuss the address. Index and serial number can be retrieved using the address and you can make the LED blink for physical identification purposes. More importantly, almost all other calls - getting, setting measurement attributes and making measurements - require the address.

There are more than 80 functions requiring the address. A few of the more commonly used functions are listed below:

- System/Sensor Management Calls
    - LB_ChangeAddress – changes the address from its current value to a new value
    - LB_WillAddressConflict – returns a 1 if the address passed to the function will cause an address conflict
    - LB_IsSensorConnected_Addr – indicates if a sensor with the address of interest is connected to the system
    - LB_GetSerNo_Addr – gets the serial number of the sensor with the address of interest
    - LB_InitializeSensor_Addr – initialzes the sensor
    - LB_BlinkLED_Addr – blinks the LED (useful in identifying the units physically)
- Measurement Calls
    - LB_MeasureCW – makes a CW measurement
    - LB_MeasurePulse – makes a pulse measurement. Returns pulse power, peak power, average power and duty cycle
- Basic Measurment Properties
    - LB_SetFrequency – sets the frequency (Hz)
    - LB_GetFrequency – gets the frequency (Hz)
    - LB_SetAverages – gets the number of averages
    - LB_GetAverages – sets the number of averages
    - LB_SetMeasurementPowerUnits – sets the measurement units to dBm, dBW, dBkW, dBuV, V or W
    - LB_GetMeasurementPowerUnits – gets the measurement units

Just a few of the calls are listed here because there are an additional 50-70 calls. This section is concerned with “management” calls as they represent a small percentage of all the calls. See the guide for additional calls and details.

Function Calls for The Pulse Profiling Application – Model LB480A & LB680A

The following functions are exported from a Visual C++ 2005 project. The calling convention used is _stdcall. The declarations are available in various examples for C SHARP, VB 6.0 and VB.NET.

LIBRARY "LB_API2"

The LB_API2.obj and LB_API2.h files are provided in the application directory for those that use C++ and the _stdcall calling convention has been retained. Please give us a call If a different calling convention is required. We may be able to supply it on a case by case basis.

The driver is installed using a .inf file. The supplied LB4XX_2K.inf file is in the Ladybug Technologies LLC\LB480A or Pulse Profiling sub-directory of the install directory.

The declarations for the various programming environments are in the application directory and in various sub-directories. These files include the type or structure declarations and some useful constants. The files are named as follows:

VB 6.0	modLBDeclarations.vb
C SHARP	LB2_Declarations.cs
VB.NET	LB_Declarations.vb

Finally, we encourage you to look at the examples provided. Time spent looking at these examples will likely answer a number of your questions.

NOTE: These routines assume the user understands and is familiar with the notion that pre-allocated buffers are often required. This is especially required when strings (such as serial number) or arrays are being passed back from the driver by reference (or pointer).

PP_AnalysisTraceIsValid

Description: Checks to ensure that the current analysis trace is valid. If the analysis trace is valid a 1 is returned. If it is not valid a 0 or less is returned. Note that all measurements, gates and marker functions operate on the analysis trace. An analysis trace is obtained most commonly by calling `PP_CurrTrace2AnalysisTrace` after having taken a trace (see `PP_GetTrace`).

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_AnalysisTraceIsValid(long  addr);
```

VB.NET

```
Public Declare Function PP_AnalysisTraceIsValid Lib "LB_API2.dll" _
      (ByRef addr As Integer) _
      As Integer
```

C SHARP

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_AnalysisTraceIsValid(int addr);
```

PP_CheckTrigger

Description: Checks the trigger source for an active trigger. If a trigger is detected a value > 0 is returned. If a trigger is not detected a value <= 0 is returned.

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_CheckTrigger(long addr);
```

VB.NET

```
Public Declare Function PP_CheckTrigger Lib "LB_API2.dll" _
      (ByVal addr As Integer) As Integer
```

C SHARP

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_CheckTrigger(long addr);
```

PP_CnvtTrace

Description: Converts a trace (trIn) from one unit to another and stores the converted values in a new trace (trOut). The power unit values units are shown below in the enumeration. The valid values are 0..7 (dBm…V) . Note that units may not be DBREL (dB relative) or a value of 8.

```
enum PWR_UNITS
{
	DBM = 0,		// dBm
	DBW = 1,		// dBW
	DBKW = 2,		// dBkW
	DBUV = 3,		// dBuV
	DBMV = 4,		// dBmV
	DBV = 5,		// dBV
	W = 6,			// Watts
	V = 7,			// Volts
	DBREL = 8		// dB Relative
};
```

Prototype:

Pass Parameters:

addr – address of the selected sensor

*trIn – a pointer to an array of doubles (user must allocate the array) that will be converted. This is the source data.

trLen – a 32 bit integer indicating the length of the array

*trOut - a pointer to an array of doubles (user must allocate the array) that will contain the converted data. This is the destination data.

pwrUnitsIn – power units of the source data

pwrUnitsOut – power units of the destination data

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_CnvtTrace(long addr, double* trIn, long trLen, double* trOut, long
pwrUnitsIn, long pwrUnitsOut);
```

VB.NET

```
Public Declare Function PP_CnvtTrace Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByRef trIn As Double, _
	ByVal trLen As Integer, _
	ByRef trOut As Double, _
```

```
      ByVal pwrUnitsIn As PWR_UNITS, _
      ByVal pwrUnitsOut As PWR_UNITS) _
      As Integer
```

C SHARP

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_CnvtTrace
      (int addr,
      ref double trIn,
      int trLen,
      ref double trOut,
      int pwrUnitsIn,
      int pwrUnitsOut);
```

PP_CurrTrace2AnalysisTrace

Description: The driver potentially holds 2 traces for each initialized sensor. One trace is the current trace. Then second trace is the analysis trace. The current trace is the most recently taken trace. The analysis trace is the trace data used to make measurements. This call copies the current trace to the analysis trace and returns a copy of that trace.

Pass Parameters:

addr – address of the selected sensor

*tr – pointer to an array of doubles

trLen – the length of the trace

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_CurrTrace2AnalysisTrace(long addr, double*tr, long trLen);
```

VB.NET

```
Public Declare Function PP_CurrTrace2AnalysisTrace Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef tr As Double, _
      ByVal trLen As Integer) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_CurrTrace2AnalysisTrace
      (int addr,
      ref double tr,
      int trLen);
```

PP_GatePositionIsValid

Description: Determines of the specified gate is valid. The gate index may be 0..4. For the gate to be valid the following conditions must exist:

- A valid analysis trace must exist
- The gate state must be on
- The left and right sides of the gate must be positioned within the boundaries of the current analysis trace.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the gate

*valid – pointer to a 32 bit integer, if the return value > 0 then the gate position is valid. If valid is <= 0 the gate position is not valid.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GatePositionIsValid(long addr, long gateIdx, long* valid);
```

VB.NET

```
Public Declare Function PP_GatePositionIsValid Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef valid As Integer)
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GatePositionIsValid
      (int addr,
      int gateIdx,
      ref int valid);
```

PP_SetAvgMode

PP_SetAvgMode
PP_GetAvgMode
PP_SetTraceAvgs
PP_GetTraceAvgs
PP_ResetTraceAveraging

Description: Trace averaging can be very important to making good measurements. In any case trace averaging will reduce the noise on the trace. There are three elements to trace averaging. First is setting the mode. Second is selecting the number of traces to average. The final element is controlling the current state of averaging.

PP_Set(Get)AvgMode sets or gets the current trace averaging mode. The averaging mode may be off, auto-reset or manual reset. If averaging mode is off then averaging will not be done. If it is auto reset then when the auto reset criteria is satisfied trace averaging will restart (all old averages will be thrown away). If averaging mode is manual reset then the averaging will continue until a call is made to change the averages, turn the averaging off or until the call to reset the averaging is made.

PP_Set(Get)TraceAvgs determines the number of traces that are averaged. This number may be between 1 and 100. Finally, PP_ResetTraceAveraging restarts the averaging process with the next trace if the mode is auto reset or manual reset.

Pass Parameters:

addr – address of the selected sensor

*mode – pointer to AVG_MODE (32 bit integer)

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum AVG_MODE
{
      AVG_OFF = 0,
      AVG_AUTO_RESET = 1,
      AVG_MANUAL_RESET = 2,
}

long __stdcall PP_GetAvgMode(long addr, AVG_MODE *mode);
long __stdcall PP_SetAvgMode(long addr, AVG_MODE mode);
long __stdcall PP_SetTraceAvgs(long addr, long averages);
long __stdcall PP_GetTraceAvgs(long addr, long*averages);
long __stdcall PP_ResetTraceAveraging(long addr);
```

VB.NET

```
Public Enum AVG_MODE
      AVG_OFF = 0
      AVG_AUTO_RESET = 1
```

```
      AVG_MANUAL_RESET = 2
End Enum

Public Declare Function PP_GetAvgMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef AvgMode As Integer) _
      As Integer
Public Declare Function PP_SetAvgMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal AvgMode As Integer) _
      As Integer
Public Declare Function PP_SetTraceAvgs Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal Avgs As Integer) _
      As Integer
Public Declare Function PP_GetTraceAvgs Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef Avgs As Integer) _
      As Integer
Public Declare Function PP_ResetTraceAveraging Lib "LB_API2.dll" _
      (ByVal addr As Integer) _
      As Integer
```

C Sharp

```
public enum AVG_MODE
{
      AVG_OFF = 0,
      AVG_AUTO_RESET = 1,
      AVG_MANUAL_RESET = 2,
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetAvgMode(int addr, ref AVG_MODE mode);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetAvgMode(int addr, AVG_MODE mode);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTraceAvgs(int addr, int averages);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTraceAvgs(int addr, ref int  averages);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_ResetTraceAveraging(int addr);
```

PP_SetAvgResetSens (and related calls)

PP_SetAvgResetSens
PP_GetAvgResetSens

Description: This gets or sets the criteria used to reset the averaging when the averaging mode is `AVG_AUTO_RESET` (see PP_GetAvgMode and PP_SetAvgMode).

Pass Parameters:

addr – address of the selected sensor

*sensitivity – value change required in dB before auto reset is satisfied

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetAvgResetSens(long addr, double* sensitivity);
long __stdcall PP_SetAvgResetSens(long addr, double sensitivity);
```

VB.NET

```
Public Declare Function PP_SetAvgResetSens Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal ResetSensitivity As Double) _
      As Integer

Public Declare Function PP_GetAvgResetSens Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef ResetSensitivity As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetAvgResetSens(int addr, double sensitivity);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetAvgResetSens(int addr, ref double sensitivity);
```

PP_SetFilter (and related calls)

PP_SetFilter
PP_GetFilter
PP_SetPoles
PP_GetPoles

Description: Sets or returns the enum associated with the current filter settings. Note that to be able to vary the filter setting above 100kHz the sensor must have option 004 installed (filter options). The enum for the various filter poles corner frequencies are shown below. The poles vary the slope of the filter skirt while the cutoff varies the 3dB point of the filter.

Pass Parameters:

addr – address of the selected sensor
fltrIdx – index of cutoff frequency
fltrPolse – index of filter poles

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum FLT_POLES
{
	ONE_POLE = 0,
	TWO_POLES = 1,
	FOUR_POLES = 2
};

enum FLT_CO_FREQ
{
	FLT_UNK  = -1,		// filter unknown
	FLT_DIS  = 0,		// filters disabled
	FLT_100K = 1,		// 100KHz
	FLT_200K = 2,		// 200KHz
	FLT_300K = 3,		// 300KHz
	FLT_500K = 4,		// 500KHz
	FLT_1M = 5,		// 1MHz
	FLT_2M = 6,		// 2MHz
	FLT_3M = 7,		// 3MHz
	FLT_5M = 8,		// 5MHz
	FLT_MAX = 9		// >10MHz
};

long __stdcall PP_GetFilter(long addr, FLT_CO_FREQ* fltrIdx);
long __stdcall PP_SetFilter(long addr, FLT_CO_FREQ fltrIdx);
long __stdcall PP_SetPoles(long addr, FLT_POLES fltrPoles);
long __stdcall PP_GetPoles(long addr, FLT_POLES* fltrPoles);
```

VB.NET

```
Public Enum FLT_POLES
	ONE_POLE = 0
	TWO_POLES = 1
```

```
      FOUR_POLES = 2
End Enum

Public Enum FLT_CO_FREQ
      FLT_UNK  = -1           'filter unknown
      FLT_DIS  = 0            'filters disabled
      FLT_100K = 1            '100KHz
      FLT_200K = 2            '200KHz
      FLT_300K = 3            '300KHz
      FLT_500K = 4            '500KHz
      FLT_1M = 5              '1MHz
      FLT_2M = 6              '2MHz
      FLT_3M = 7              '3MHz
      FLT_5M = 8              '5MHz
      FLT_MAX = 9             '>10MHz
End Enum

Public Declare Function PP_GetFilter Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef fltrIdx As Integer) _
      As Integer

Public Declare Function PP_SetFilter Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal fltrIdx As Integer) _
      As Integer

Public Declare Function PP_GetPoles Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef fltrPoles As Integer) _
      As Integer

Public Declare Function PP_SetPoles Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal fltrPoles As Integer) _
      As Integer
```

C Sharp

```
public enum FLT_POLES
{
      ONE_POLE = 0,
      TWO_POLES = 1,
      FOUR_POLES = 2
};

public enum FLT_CO_FREQ
{
      FLT_UNK = -1,           // filter unknown
      FLT_DIS = 0,            // filters disabled
      FLT_100K = 1,           // 100KHz
      FLT_200K = 2,           // 200KHz
      FLT_300K = 3,           // 300KHz
      FLT_500K = 4,           // 500KHz
      FLT_1M = 5,             // 1MHz
      FLT_2M = 6,             // 2MHz
      FLT_3M = 7,             // 3MHz
```

```
        FLT_5M = 8,             // 5MHz
        FLT_MAX = 9             // >=10MHz
};

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetPoles(int addr, FLT_POLES fltrPoles);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPoles(int addr, ref FLT_POLES fltrPoles);
```

PP_GetGateAveragePower

Description: Returns the average power of the span defined in the analysis trace specified by the gate.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*avgPwr – returns the average power between the gate edges.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateAveragePower(long addr, long gateIdx, double* avgPwr);
```

VB.NET

```
Public Declare Function PP_GetGateAveragePower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef avgPwr As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateAveragePower
      (int addr,
      int gateIdx,
      ref double avgPwr);
```

PP_GetGateCrestFactor

Description : Returns the create factor (in dB) of the span in the analysis trace specified by the gate.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*crFactor – returns the crest factor in dB (peak power – average power) between the gate edges

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateCrestFactor(long addr, long gateIdx, double* crFactor);
```

VB.NET

```
Public Declare Function PP_GetGateCrestFactor Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef crFactor As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateCrestFactor
      (int addr,
      int gateIdx,
      ref double crFactor);
```

PP_GetGateDroop

Description: : Returns the droop of the span in the analysis trace specified by the gate. The droop will be the difference in power between the area at beginning and end of the gate edges.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*droop – returns droop of the signal in dB. This assumes that the gate edges are appropriately positioned (near the beginning and end edges of a pulse). It returns the difference between the first 5% and the last 5% of the area defined by the gate.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateDroop(long addr, long gateIdx, double* droop);
```

VB.NET

```
Public Declare Function PP_GetGateDroop Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef droop As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateDroop
      (int addr,
      int gateIdx,
      ref double droop);
```

PP_GetGateDutyCycle

Description: Returns the duty cycle (as a decimal) of span in the analysis trace specified by the gate.

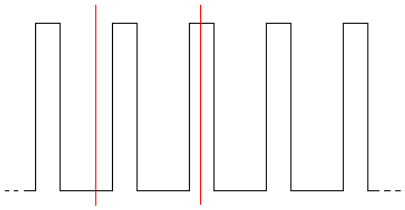
Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*dutyCycle – returns the ratio of on time to off time. The gate edges may contain many pulses. However, it must contain at least one full pulse (including the rising edge) followed by the rising edge of the a second pulse. If the gate contains multiple pulses the first full cycle will be used to make the measurement. The value returned is a decimal value. Multiply by 100 to convert to percent.

The diagram below depicts the minimum span defined by the gate edges for a proper duty cycle measurement. The gate edges are shown in red.



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateDutyCycle(long addr, long gateIdx, double* dutyCycle);
```

VB.NET

```
Public Declare Function PP_GetGateDroop Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef droop As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateDutyCycle
      (int addr,
      int gateIdx,
      ref double dutyCycle);
```

PP_GetGateEndPosition

Description: Returns the location, as an index in the analysis trace, of the right side of the specified gate.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*trIdx – returns the trace index (assuming a zero based array) of the right or ending side of the gate. The trace referred to here is a trace the analysis trace. trace (see PP_CurrTrace2AnalysisTrace）.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateEndPosition(long addr, long gateIdx, long* trIdx);
```

VB.NET

```
Public Declare Function PP_GetGateEndPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef trIdx As Integer) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateEndPosition
      (int addr,
      int gateIdx,
      ref int  trIdx);
```

PP_GetGateFallTime

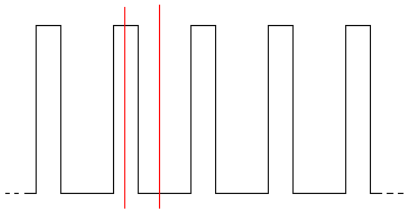
Description: Returns the fall time in microseconds of the pulse delineated by the selected gate. The gate must be properly positioned to return a proper value. The left side of the gate must be positioned between a pulse rising and falling edge. The right side must be positioned after the next falling edge.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate, gate mode must be on (see PP_GetGateMode) and the position of the gate edges must be valid (see PP_GatePositionIsValid)

*gateTm – returns the position in microseconds of the right or ending side of the gate referenced to the beginning of the trace.. The trace referred to is the analysis trace. The diagram below depicts the minimum span of the analysis trace that must be defined by the gate. The gate edges are shown in red.



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateFallTime(long addr, long gateIdx, double* fallTm);
```

VB.NET

```
Public Declare Function PP_GetGateFallTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef fallTm As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateFallTime
      (int addr,
      int gateIdx,
      ref double fallTm);
```

PP_SetGateMode (and related calls)

PP_SetGateMode
PP_GetGateMode

Description: Sets or gets the gate mode. The gate mode must be on to position the gate edges or use the gate for measurements.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*mode – returns the mode of the gate. A gate must be in the GATE_ON mode to position the gate edges and to make measurements.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum GATE_MODE
{
    GATE_OFF = 0,
      GATE_ON = 1
};

long __stdcall PP_GetGateMode(long addr, long gateIdx, GATE_MODE * mode);
long __stdcall PP_SetGateMode(long addr, long gateIdx, GATE_MODE mode);
```

VB.NET

```
Public Enum GATE_MODE
      GATE_OFF = 0
      GATE_ON = 1
End Enum

Public Declare Function PP_SetGateMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mrkIdx As Integer, _
      ByVal mode As Integer) _
      As Integer
Public Declare Function PP_GetGateMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef gateMode As Integer) _
      As Integer
```

C Sharp

```
public enum GATE_MODE
```

```
{
      GATE_OFF = 0,
      GATE_ON = 1
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateMode
      (int addr,
      int gateIdx,
      GATE_MODE mode);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateMode
      (int addr,
      int gateIdx,
      ref GATE_MODE mode);
```

PP_GetGateOverShoot

Description: Returns the overshoot in dB. Overshoot is calculated using the following process:

- Span defined by the gate (gateIdx) is broken into two regions:
  - First quarter
  - Last three quarters
- Find the peak in first quarter of the span
- Find the average of last three quarters of the span
- Return the difference between the peak in the first and the average of the last three quarters

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*overShoot –overshoot in dB as outlined above

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateOverShoot(long addr, long gateIdx, double* overShoot);
```

VB.NET

```
Public Declare Function PP_GetGateOverShoot Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef overShoot As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateOverShoot
      (int addr,
      int gateIdx,
      ref double overShoot);
```

PP_GetGatePeakPower

Description: Returns the peak power measured of the analysis trace as defined by the gate edges.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*pkPwr – returns the peak power in dB. The gate must be on and have a valid position in the analysis trace. The edges of the gate need not contain a pulse.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGatePeakPower(long addr, long gateIdx, double* pkPwr);
```

VB.NET

```
Public Declare Function PP_GetGatePeakPower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef pkPwr As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePeakPower
      (int addr,
      int gateIdx,
      ref double pkPwr);
```

PP_GetGatePRF

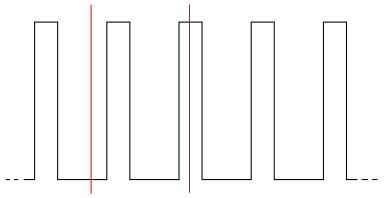
Description: Returns the PRF or pulse repetition frequency in Hertz as defined by the inverse of the time between the rising edges of the first two complete pulses present in the span defined by the gate (gateIdx). A complete pulse is a rising edge followed by falling edge. If two complete pulses are not present in the span defined by the gate an error (<0 is returned).

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*PRFreg – returns the frequency in Hertz. The span defined by the gate must contain at least one complete pulse followed by the rising edge of the next pulse. The PRF is measured from rising edge to rising edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate. The gate must be on and the analysis trace must be valid. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. However, the first two rising edges will be used to make the measurement.



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGatePRF(long addr, long gateIdx, double* PRFreq);
```

VB.NET

```
Public Declare Function PP_GetGatePRF Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef PRFreq As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePRF
      (int addr,
      int gateIdx,
      ref double PRFreq);
```

PP_GetGatePRT

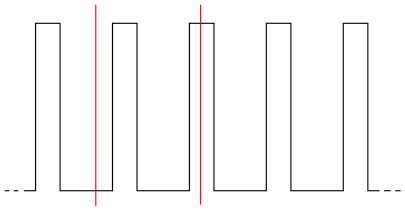
Description: Returns the PRT or pulse repetition time in microseconds using the same algorithm defined for PRF. The sole difference is that time instead of frequency is returned.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*PRTime – returns the time in microseconds. The span defined by the gate must contain at least one complete pulse followed by the rising edge of the next pulse. The PRT is measured from rising edge to rising edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate. The gate must be on and the analysis trace must be valid. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. However, the first two rising edges will be used to make the measurement.



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGatePRT(long addr, long gateIdx, double* PRTime);
```

VB.NET

```
Public Declare Function PP_GetGatePRF Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef PRFreq As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePRF
      (int addr,
      int gateIdx,
      ref double PRFreq);
```

PP_GetGatePulseWidth

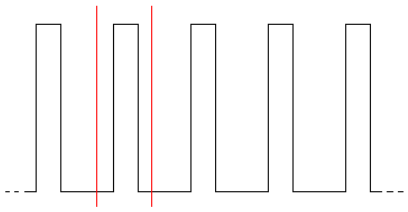
Description: Measures the pulse width in microseconds.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*plsWidth – returns pulse width in microseconds. The span defined by the gate must contain at least one complete pulse. Specifically it must include a rising edge followed by a falling edge. The pulse width is measured from rising edge to the subsequent falling edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate. The gate must be on and the analysis trace must be valid. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. However, the first complete pulse will be used to make the measurement



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGatePulseWidth(long addr, long gateIdx, double* plsWidth);
```

VB.NET

```
Public Declare Function PP_GetGatePulseWidth Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef plsWidth As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePulseWidth
      (int addr,
      int gateIdx,
      ref double plsWidth);
```

PP_GetGatePulsePower

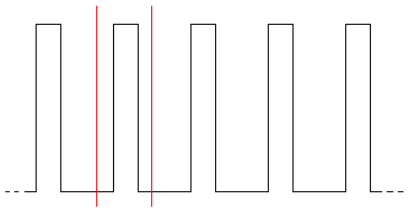
Description: Returns average pulse power.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*plsPwr – returns pulse power in dBm. The span defined by the gate must contain at least one complete pulse. Specifically it must include a rising edge followed by a falling edge. The average pulse power is measured by averaging all of the sample between the rising edge to the subsequent falling edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate. The gate must be on and the analysis trace must be valid. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. However, the first complete pulse will be used to make the measurement



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGatePulsePower(long addr, long gateIdx, double* plsPwr);
```

VB.NET

```
Public Declare Function PP_GetGatePulsePower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef plsPwr As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePulsePower
      (int addr,
      int gateIdx,
      ref double plsPwr);
```

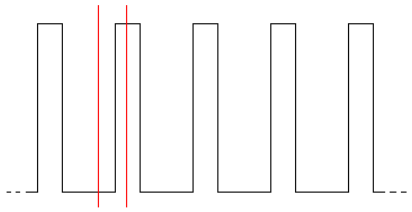
PP_GetGateRiseTime

Description: Returns rise time in microseconds.

Pass Parameters:

addr – address of the selected sensor

*riseTm – Measured rise time in microseconds. The gate edges must be set as shown below.



Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetGateRiseTime(long addr, long gateIdx, double* riseTm);
```

VB.NET

```
Public Declare Function PP_GetGateRiseTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef riseTm As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateRiseTime
      (int addr,
      int gateIdx,
      ref double riseTm);
```

PP_SetGateStartEndPosition (and related calls)

PP_SetGateStartEndPosition
PP_GetGateStartEndPosition
PP_SetGateStartEndTime
PP_GetGateStartEndTime
PP_SetGateStartPosition
PP_GetGateStartPosition
PP_SetGateEndPosition
PP_GetGateEndPosition
PP_SetGateStartTime
PP_GetGateStartTime
PP_SetGateEndTime
PP_GetGateEndTime

Description: Sets or gets the gate start (left side) and/or end (right side) in terms of trace index or time. If the index or time out of range (i.e. index or time < 0 or index > trace length or time > sweep time) then the gate position will be reported as invalid. Time is in microseconds. Index is trace index.

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the desired gate (0..4)

sttIdx or sttTm – start or left side of the gate as an index into the trace (sttIdx < stpIdx)

stpIdx or endTm – stop or right side of the gate as an index into the trace (stpIdx > sttIdx)

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetGateStartEndPosition(long addr,
                                          long gateIdx,
                                          long sttIdx,
                                          long endIdx);
long __stdcall PP_GetGateStartEndPosition(long addr,
                                          long gateIdx,
                                          long* trSttIdx,
                                          long* trEndIdx);
long __stdcall PP_SetGateStartEndTime(long addr,
                                          long gateIdx,
                                          double sttTm,
                                          double endTm);
long __stdcall PP_GetGateStartEndTime(long addr,
                                          long gateIdx,
                                          double* sttTm,
                                          double* endTm);

long __stdcall PP_SetGateStartPosition(long addr,long gateIdx,long trSttIdx);
```

```
long __stdcall PP_GetGateStartPosition(long addr,long gateIdx,long* trSttIdx);
long __stdcall PP_SetGateStartTime(long addr,long gateIdx,double sttTm);
long __stdcall PP_GetGateStartTime(long addr,long gateIdx,double* sttTm);

long __stdcall PP_SetGateEndPosition(long addr, long gateIdx, long trIdx);
long __stdcall PP_GetGateEndPosition(long addr,long gateIdx,long* trEndIdx);
long __stdcall PP_SetGateEndTime(long addr, long gateIdx, double endTm);
long __stdcall PP_GetGateEndTime (long addr,long gateIdx,double* endTm);
```

VB.NET

```
Public Declare Function PP_SetGateStartEndPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByVal trSttIdx As Integer, _
      ByVal trEndIdx As Integer) As Integer
Public Declare Function PP_GetGateStartEndPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef trSttIdx As Integer, _
      ByRef trEndIdx As Integer) _
      As Integer
Public Declare Function PP_SetGateStartEndTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByVal sttTm As Double, _
      ByVal endTm As Double) _
      As Integer
Public Declare Function PP_GetGateStartEndTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gateIdx As Integer, _
      ByRef sttTm As Double, _
      ByRef endTm As Double) _
      As Integer

Public Declare Function PP_SetGateStartPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByVal trSttIdx As Integer) As Integer
Public Declare Function PP_GetGateStartPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByRef trSttIdx As Integer) As Integer
Public Declare Function PP_SetGateStartTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByVal sttTm As Double) As Integer
Public Declare Function PP_GetGateStartTimeLib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByRef sttTm As Double) As Integer

Public Declare Function PP_SetGateEndPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal trEndIdx As Integer, _
      ByVal trSttIdx As Integer) As Integer
Public Declare Function PP_GetGateEndPosition Lib "LB_API2.dll" _
```

```
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByRef trEndIdx As Integer) As Integer
Public Declare Function PP_SetGateEndTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByVal endTm As Double) As Integer
Public Declare Function PP_GetGateEndTimeLib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal gatIdx As Integer, _
      ByRef endTm As Double) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateStartEndPosition
      (int addr,
      int gateIdx,
      int trSttIdx,
      int trEndIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateStartEndPosition
      (int addr,
      int gateIdx,
      ref int trSttIdx,
      ref int  trEndIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateStartEndTime
      (int addr,
      int gateIdx,
      double sttTm,
      double endTm);
 [System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateStartEndTime
      (int addr,
      int gateIdx,
      ref double gateSttTm,
      ref double gateEndTm);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateStartPosition
      (int addr,
      int gateIdx,
      int trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateStartPosition
      (int addr,
      int gateIdx,
      ref int  trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateStartTime
      (int addr,
      int gateIdx,
      double gateTm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateStartTime
      (int addr,
      int gateIdx,
```

```
      ref double gateTm);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateEndPosition
      (int addr,
      int gateIdx,
      int trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateEndPosition
      (int addr,
      int gateIdx,
      ref int  trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetGateEndTime
      (int addr,
      int gateIdx,
      double gateTm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGateEndTime
      (int addr,
      int gateIdx,
      ref double gateTm);
```

PP_GetMarkerAmp

Description: Returns the amplitude of the trace at the point indicated by the marker.

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker (0..4)

*mkrAmp – amplitude (in dBm) of the position indicated by the marker.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetMarkerAmp(long addr, long mrkIdx, double* mkrAmp);
```

VB.NET

```
Public Declare Function PP_GetMarkerAmp Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef mkrAmp As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerAmp
      (int addr,
      int mrkIdx,
      ref double mkrAmp);
```

PP_GetMarkerDeltaAmp

Description: Returns the difference in amplitude between the normal marker and the delta marker in dBm.

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetMarkerDeltaAmp(long addr, long mrkIdx, double* deltaMkrAmp);
```

VB.NET

```
Public Declare Function PP_GetMarkerDeltaAmp Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef deltaMkrAmp As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerDeltaAmp
      (int addr,
      int mrkIdx,
      ref double deltaMkrAmp);
```

PP_SetMarkerDeltaTime (and related calls)

PP_SetMarkerDeltaTime
PP_GetMarkerDeltaTime

Description: Sets or gets the positions the selected marker in microseconds.

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker (0..4)

mrkTm – time in microseconds

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetMarkerDeltaTime(long addr, long mrkIdx, double mkrTm);
long __stdcall PP_GetMarkerDeltaTime(long addr, long mrkIdx, double* mkrTm);
```

VB.NET

```
Public Declare Function PP_SetMarkerDeltaTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByVal mkrTm As Double) _
      As Integer
Public Declare Function PP_GetMarkerDeltaTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef mkrTm As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetMarkerDeltaTime
      (int addr,
      int mrkIdx,
      double mkrTm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerDeltaTime
      (int addr,
      int mrkIdx,
      ref double mkrTm);
```

PP_SetMarkerMode (and related calls)

PP_SetMarkerMode
PP_GetMarkerMode

Description: Sets the marker mode to on, normal or delta marker.

Pass Parameters:

addr – address of the selected sensor

mrkIdx – marker index (0..4)

mode – marker mode is off, normal or delta. If the marker is in normal mode. In normal mode the normal marker is be positioned or measured. If the marker is in delta mode then the delta marker is positioned or measured.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum MARKER_MODE
{
	MKR_OFF = 0,
	NORMAL_MKR = 1,
	DELTA_MKR = 2
};

long __stdcall PP_SetMarkerMode(long addr, long mrkIdx, MARKER_MODE mode);
long __stdcall PP_GetMarkerMode(long addr, long mrkIdx, MARKER_MODE * mode);
```

VB.NET

```
Public Enum MARKER_MODE
	MKR_OFF = 0
	NORMAL_MKR = 1
	DELTA_MKR = 2
End Enum

Public Declare Function PP_SetMarkerMode Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByVal mrkIdx As Integer, _
	ByVal mode As Integer) _
	As Integer
Public Declare Function PP_GetMarkerMode Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByVal mkrIdx As Integer, _
	ByRef mode As Integer) _
	As Integer
```

C Sharp

```
public enum MARKER_MODE
{
      MKR_OFF = 0,
      NORMAL_MKR = 1,
      DELTA_MKR = 2
}
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetMarkerMode
      (int addr,
      int mrkIdx,
      MARKER_MODE mode);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerMode
      (int addr,
      int mrkIdx,
      ref MARKER_MODE mode);
```

PP_SetMarkerPosition (and related calls)

PP_SetMarkerPosition
PP_GetMarkerPosition
PP_SetMarkerPositionTime
PP_GetMarkerPositionTime

Description: Sets or gets the position of the normal or delta marker depending on the marker mode. If the marker is in normal mode then the normal marker is positioned. If the marker is in delta mode then the delta marker is positioned and the normal marker is unaffected. The marker may be positioned in terms of index or time (microseconds).

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker

trIdx or mrkTm –trace index or time in microseconds

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetMarkerPosition(long addr, long mrkIdx, long trIdx);
long __stdcall PP_GetMarkerPosition(long addr, long mrkIdx, long* trIdx);
long __stdcall PP_SetMarkerPositionTime(long addr, long mrkIdx, double mkrTm);
long __stdcall PP_GetMarkerPositionTime(long addr, long mrkIdx, double* mkrTm);
```

VB.NET

```
Public Declare Function PP_SetMarkerPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByVal trIdx As Integer) _
      As Integer
Public Declare Function PP_GetMarkerPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef trIdx As Integer) _
      As Integer
Public Declare Function PP_SetMarkerPositionTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByVal mkrTm As Double) _
      As Integer
Public Declare Function PP_GetMarkerPositionTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef mkrTm As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetMarkerPosition
      (int addr,
      int mrkIdx,
      int trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerPosition
      (int addr,
      int mrkIdx,
      ref int  trIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetMarkerPositionTime
      (int addr,
      int mrkIdx,
      double mkrTm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMarkerPositionTime
      (int addr,
      int mrkIdx,
      ref double mkrTm);
```

PP_SetMeasurementThreshold (and related calls)

PP_SetMeasurementThreshold
PP_GetMeasurementThreshold

Description: Sets or gets the measurement threshold. The measurement threshold, along with peak criteria affects a number of measurement routines. Most notably is the peak routines. In short, the threshold sets the lowest value considered in the trace. When a trace is searched for peaks (the analysis trace) before the search takes place all trace values lower than the threshold are set equal to the threshold. Then the trace is searched for peaks. The threshold is set or reported in dBm.

In general the threshold should be regarded as a filter.

Pass Parameters:

addr – address of the selected sensor

measThreshold_dBm – measurement threshold in dBm

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetMeasurementThreshold(long addr, double measThreshold_dBm);
long __stdcall PP_GetMeasurementThreshold(long addr, double* measThreshold_dBm);
```

VB.NET

```
Public Declare Function PP_GetMeasurementThreshold Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef measThreshold_dBm As Double) _
      As Integer
Public Declare Function PP_SetMeasurementThreshold Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal measThreshold_dBm As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetMeasurementThreshold
      (int addr,
      double measThreshold_dBm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetMeasurementThreshold
      (int addr,
      ref double measThreshold_dBm);
```

PP_GetPeaks_Val (and related calls)

PP_GetPeaks_Val
PP_GetPeaks_Idx
PP_GetPeaksFromTr_Val
PP_GetPeaksFromTr_Idx
PP_GetPeaks_VEE_Idx
PP_GetPeaks_VEE_Val

Description: Returns a set of peaks from either the analysis trce (PP_GetPeaks_Val and PP_GetPeaks_Idx) or from a trace passed to the routine. The more complex routines have the added advantage that the trace may be any compatible trace and can use a peak criteria and threshold different to the values currently set.

The peaks returned are ordered by index (left to right in the trace) or by value (highest to lowest). In all cases the user must allocate an array sufficiently large to hold largest number of peaks. A safe array size is half the length of the trace (see the PP_SetSweepTime). This is safe because a rise and fall is required to identify a peak. This means that at a minimum of two points or pixels is required for each peak.

Depending on your development environment you may choose to use the _VEE calls. These calls do not use the Peak structure. Instead they flatten out the array of structures into an array of long and doubles. This is used in environments such as Agilent VEE and National Instruments LabView. Other environments may find these calls more suitable.

Pass Parameters:

addr – address of the selected sensor

peak – an array of peaks (see the structure definition)

maxPks – number of peaks allocated (indicates the size of the peaks array allocated by the user)

pksUsed – number of peaks found or used by the peaks routine.

peakCrit – peak criteria used to define a peak.

measThresh – measurement threshold use d to filter peaks.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
struct Peak
{
      long trIdx;
      double value;
};

long __stdcall PP_GetPeaks_Val(long addr, Peak* peaks, long maxPks,long* pksUsed);
long __stdcall PP_GetPeaks_Idx(long addr,Peak* peaks,long maxPks,long* pksUsed);
long __stdcall PP_GetPeaks_VEE_Idx(long addr,
                              long* pkIndicies,
```

```
                                double* pkValues,
                                long maxPks,
                                long* pksUsed);
long __stdcall PP_GetPeaks_VEE_Val(long addr,
                                long* pkIndicies,
                                double* pkValues,
                                long maxPks,
                                long* pksUsed);
long __stdcall PP_GetPeaksFromTr_Val(double* tr,
                                long trLen,
                                long units,
                                double peakCrit,
                                double measThresh,
                                Peak* peaks,
                                long maxPks,
                                long* pksUsed);
long __stdcall PP_GetPeaksFromTr_Idx(double* tr,
                                long trLen,
                                long units,
                                double peakCrit,
                                double measThresh,
                                Peak* peaks,
                                long maxPks,
                                long* pksUsed);
```

VB.NET

```
<StructLayout(LayoutKind.Sequential, Size:=12)> _
Public Structure Peak
      Dim trIdx As Integer
      Dim value As Double
End Structure

Public Declare Function PP_GetPeaks_Idx Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef Peak peaks, _
      ByVal maxPks As Integer, _
      ByVal pksUsed As Integer) As Integer -
Public Declare Function PP_GetPeaks_Val Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef Peak peaks,  _
      ByVal maxPks As Integer, _
      ByVal pksUsed As Integer) As Integer _
Public Declare Function PP_GetPeaksFromTr_Idx Lib "LB_API2.dll" _
      (ByRef tr As Double, _
      ByVal trLen As Integer, _
      ByVal units As Integer, _
      ByVal pkCrit As Double, _
      ByVal measThresh As Double, _
      ByRef peaks As Peak, _
      ByVal maxPks As Integer, _
      ByRef pksUsed As Integer) As Integer
Public Declare Function PP_GetPeaksFromTr_Val Lib "LB_API2.dll" _
      (ByRef tr As Double, _
      ByVal trLen As Integer, _
      ByVal units As Integer, _
```

```
      ByVal pkCrit As Double, _
      ByVal measThresh As Double, _
      ByRef peaks As Peak, _
      ByVal maxPks As Integer,  _
      ByRef pksUsed As Integer) As Integer
```

C Sharp

```
public struct Peak
{
      public int trIdx; // index where peak was found
      public double value;    // value of peak
};

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPeaks_Idx
      (int addr,
      ref Peak peaks,
      int maxPks,
      ref int pksUsed);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPeaksFromTr_Idx(ref double tr,
      int trLen,
      int units,
      double pkCrit,
      double measThresh,
      ref Peak peaks,
      int maxPks,
      ref int pksUsed);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPeaks_Val
      (int addr,
      ref Peak peaks,
      int maxPks,
      ref int pksUsed);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPeaksFromTr_Val(ref double tr,
      int trLen,
      int units,
      double pkCrit,
      double measThresh,
      ref Peak peaks,
      int maxPks,
      ref int pksUsed);
```

PP_SetPoles (and related calls)

PP_SetPoles
PP_GetPoles

Description: Sets the number of poles in the current filter. As the number of poles increase the sharpness of cutoff increases. The valid indices are 0…2 indicating the number of poles between 1..4.

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum FLT_POLES
{
	ONE_POLE = 0,
	TWO_POLES = 1,
	FOUR_POLES = 2
};

long __stdcall PP_SetPoles(long addr, FLT_POLES fltrPoles);
long __stdcall PP_GetPoles(long addr, FLT_POLES* fltrPoles);
```

VB.NET

```
Public Enum FLT_POLES
	ONE_POLE = 0
	TWO_POLES = 1
	FOUR_POLES = 2
End Enum

Public Declare Function PP_SetPoles Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByVal fltrPoles As Integer) _
	As Integer
Public Declare Function PP_GetPoles Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByRef fltrPoles As Integer) _
	As Integer
```

C Sharp

```
public enum FLT_POLES
{
	ONE_POLE = 0,
	TWO_POLES = 1,
	FOUR_POLES = 2
};
```

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetPoles
      (int addr,
      FLT_POLES fltrPoles);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPoles
      (int addr,
      ref FLT_POLES fltrPoles);
```

PP_GetPulseEdgesTime (and related calls)

PP_GetPulseEdgesTime
PP_GetPulseEdgesPosition

Description: Returns the index of the leading and trailing edges of the pulse containing the peak defined by pkTime or pkIdx. These call are intended to be used with PP_GetPeak and other routines as shown below. The following algorithm applies to measuring rise time. It uses PP_GetPulseEdgesPosition but the same algorithm works with PP_GetPulseEdgesTime also. The difference of course is that everything is in time (microseconds) instead of trace index.

- Acquire a trace (`PP_GetTrace`)
- Move current trace to analysis trace (`PP_CurrTrace2AnalysisTrace`)
- Get the peaks from the trace sorted by index (`PP_GetPeaks_Idx`)
- Check that sufficient peaks exist for the desired measurements. Many measurements require at least two pulses. Two pulses requires at least two peaks. For this check the pksUsed parameter returned in the previous `PP_GetPeaks_Idx` call.
- Select the peaks of interest (pick the first peak returned in P_GetPeaks_Idx)
- Get the edges of the pulses containing the peak (`PP_GetPulseEdgesPosition`)
- Set the mode of the selected gate to ON (`PP_SetGateMode`)
- Set the edges of the gate appropriately for the measurement: (`PP_SetGateStartEndPosition`)
    - For rise time
        - Left gate edge before the rising edge
        - Right gate edge midway between the rising and falling edge s

      Example:
      Assume a 1msec sweep time (10,000 points) for a resolution of 100nsec
      Assume a 10kHz signal with a 20% duty cycle

      Assume first peak should be located between 1000 and 1200 – assume it is located at an index of 1100
    - The pulse is 200 points or pixels wide so that the pulse edges will be about:
        - Left pulse edge: 1000
        - Right pulse edge: 1200
    - Set the gate edges as follows:
        - Left side of the gate at 950 (about 50 pixesl before the rising edge)
        - Right side of the gate at 1100 (midway between rising and falling edge)

Now you can measure the rise time using `PP_GetGateRiseTime`.

NOTE: This function and the companion functions noted above are especially useful in making programmatic measurements. These functions allow for the easiest placement of gate edges.

Pass Parameters:

addr – address of the selected sensor

pkTime or pkIdx – location of the peak in microseconds or trace index

*leftSide, *leftTrIdx – returned location of the left pulse edge in time (microseconds) or trace index

*rightSide, *rightTrIdx – returned location of the right pulse edge in time (microseconds) or trace index

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetPulseEdgesTime
      (long addr,
      double pkTime,
      double* leftSide,
      double* rightSide);
long __stdcall PP_GetPulseEdgesPosition
      (long addr,
      long pkIdx,
      long *leftTrIdx,
      long *rightTrIdx);
```

VB.NET

```
Public Declare Function PP_GetPulseEdgesPosition Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal pkIdx As Integer, _
      ByRef leftTrIdx As Integer, _
      ByRef rightTrIdx As Integer) _
      As Integer
Public Declare Function PP_GetPulseEdgesTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal pkTime As Double, _
      ByRef leftSide As Double, _
      ByRef rightSide As Double) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPulseEdgesPosition
      (int addr,
      int pkIdx,
      ref int leftTrIdx,
      ref int rightTrIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetPulseEdgesTime
      (int addr,
      double pkTime,
      ref double leftSide,
      ref double rightSide);
```

PP_SetSweepDelay

PP_SetSweepDelay
PP_GetSweepDelay

Description: Sets or gets the sweep delay in microseconds. Sweep delay is the time between the trigger and the start of data acquisition. The sweep delay limitations are as follows:

Sweep Time	Max Swp Time (Under sampled)	Max Swp Time (No under sampling)
10usec to 10msec	1 <= 10msec	
20msec to 50msec	1 <= 10msec	>10msec to 999msec
100msec to 1second		>10msec to 999msec

Delay sweep is taken in one of two ways. Sweep times at 10msec and faster always use under sampling. Under sampling tends to extend the time required to acquire data. Traces taken without under sampling may result in an increase in noise at lower power levels. However, you will see an improvement in data acquisition time. Trace averaging can be used to offset this effect.

Pass Parameters:

addr – address of the selected sensor
SwpDly –delay in microseconds before data is taken. Delay is measured from the trigger edge.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetSweepDelay(long addr, long* SwpDly);
long __stdcall PP_SetSweepDelay(long addr, long SwpDly);
```

VB.NET

```
Public Declare Function PP_SetSweepDelay Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal delay As Integer) As Integer
Public Declare Function PP_GetSweepDelay Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef delay As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetSweepDelay
      (int addr,
      int SwpDly);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetSweepDelay
      (int addr,
      ref int  SwpDly);
```

PP_SetSweepDelayMode

PP_SetSweepDelayMode
PP_GetSweepDelayMode

Description: This call turns the sweep delay on or off. The sweep delay parameter remains unchanged.

Pass Parameters:

addr – address of the selected sensor

SwpDlyMode – 0=OFFf, 1 = ON

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetSweepDelayMode(long addr, long SwpDlyMode);
long __stdcall PP_GetSweepDelayMode(long addr, long* SwpDlyMode);
```

VB.NET

```
Public Declare Function PP_SetSweepDelayMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mode As Integer) As Integer
Public Declare Function PP_GetSweepDelayMode Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef mode As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetSweepDelayMode
      (int addr,
      int SwpDlyMode);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetSweepDelayMode
      (int addr,
      ref int SwpDlyMode);
```

PP_SetSweepHoldOff (and related calls)

PP_SetSweepHoldOff
PP_GetSweepHoldOff

Description: This specifies the length of time (in microseconds) to wait after a sweep or trace is taken.

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetSweepHoldOff(long addr, long SwpHOff);
long __stdcall PP_GetSweepHoldOff(long addr, long* SwpHOff);
```

VB.NET

```
Public Declare Function PP_SetSweepHoldOff Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal SwpHOff As Integer) As Integer
Public Declare Function PP_GetSweepHoldOff Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef SwpHOff As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetSweepHoldOff(int addr, int SwpHOff);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetSweepHoldOff(int addr, ref int SwpHOff);
```

PP_SetSweepTime (and related calls)

PP_SetSweepTime
PP_GetSweepTime

Description: This routine sets or gets the sweep time (in microseconds) for the next sweep taken. Sweep time is a 1, 2, 5 sequence starting with 10usec and ending with 1 second. The table below shows the relationship between sweep times points per trace, oversampling and resolution:

Sweep Time	# Trace Points	Under Sampling	Resolution (time/points)
10 usec	480	96	0.02083 usec
20 usec	960	96	0.02083 usec
50 usec	2400	96	0.02083 usec
100 usec	4800	96	0.02083 usec
200 usec	9600	96	0.02083 usec
500 usec	10,000	48	0.05000 usec
1,000 usec	10,000	24	0.10000 usec
2,000 usec	10,000	24	0.20000 usec
5,000 usec	10,000	24	0.50000 usec
10,000 usec	10,000	24	1.00000 usec
20,000 usec	10,000	12	2.00000 usec
50,000 usec	10,000	6	5.00000 usec
100,000 usec	10,000	2	10.00000 usec
200,000 usec	10,000	1	20.00000 usec
500,000 usec	10,000	1	50.00000 usec
1,000,000 usec	10,000	1	100.00000 usec

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetSweepTime(long addr, long SwpTm);
long __stdcall PP_GetSweepTime(long addr, long* SwpTm);
```

VB.NET

```
Public Declare Function PP_SetSweepTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal swpTimeUSEC As Integer) As Integer
Public Declare Function PP_GetSweepTime Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef swpTimeUSEC As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetSweepTime(int addr, int SwpTm);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetSweepTime(int addr, ref int  SwpTm);
```

PP_GetTrace

Description: Causes the sensor to take a trace and return the resultant data. The trace is an array of equally spaced (in time) samples. All values are in dBm. The user must pass the address of the sensor, an array of doubles and the length of the array. An outline of how to take a trace and make a measurement (PRF) programmatically is shown below:

- Initialize the sensor (`LB_Initialize_Addr`)
- Set the freqeuency (`LB_SetFrequency`)
- Set the sweep time (`PP_SetSweepTime`)
- Get the length of the trace (`PP_GetTraceLength`)
- Allocate an array equal to or larger than trace length
- Get a trace (PP_GetTrace)
- Move the current trace to the analysis trace (PP_`CurrTrace2AnalysisTrace)`
- Get the peaks orders by index (`PP_GetPeaks_Idx`)
- Use the first two peaks from the previous call to get pulse edges (`PP_GetPulseEdgesPosition)`
- Set the mode of the gate to ON `(PP_SetGateMode)`
- Position the left side of the gate before the leading edge of the first pulse the right side of the gate after the trailing edge of the second pulse (`SetGateStartEndPosition)`.
- Make the PRF measurement (`PP_GetGatePRF`)

Once this sequence is accomplished a number of calls on subsequent measurements can be eliminated. The most notable is initialization. And, calls as setting frequency, sweep time, gate mode and other calls need not be made unless the state of the measurement changes. Taking this to an extreme the following sequence would repeat the same measurement (assuming no changes)

- Get a trace (PP_GetTrace)
- Move the current trace to the analysis trace (PP_`CurrTrace2AnalysisTrace)`
- Make the PRF measurement (`PP_GetGatePRF`)

This short sequence makes several assumptions. Primary among these assumptions is that the signal is very stable. However, such approaches have been used to take the average of several measurements. Another technique is to make several measurements on a single analysis trace. The sequence might look like this:

- ….
- Get the length of the trace (`PP_GetTraceLength`)
- Allocate an array equal to or larger than trace length
- Get a trace (PP_GetTrace)
- Move the current trace to the analysis trace (PP_`CurrTrace2AnalysisTrace)`
- Get the peaks orders by index (`PP_GetPeaks_Idx`)
- Use the first two peaks from the previous call to get pulse edges (`PP_GetPulseEdgesPosition)`
- Set the mode of the gate to ON `(PP_SetGateMode)`
- Position the left side of the gate before the leading edge of the first pulse the right side of the gate after the trailing edge of the second pulse (`SetGateStartEndPosition)`.
- Make the PRF measurement (`PP_GetGatePRF`)
- Using the current gate and trace make a PRT measurement (`PP_GetGatePRT`)
- Using the current gate and trace make a pulse width measurement (`PP_GetGatePulseWidth`)
- Using the same pulse edge information reposition the gate edges for a rise time measurement (`SetGateStartEndPosition)`.
- Make a rise time measurement (`PP_GetGateRiseTime`)
- …and so on

A useful subroutine might be appear as shown below:

- Start
- Set the freqeuency (`LB_SetFrequency`)
- Set the sweep time (`PP_SetSweepTime`)
- Get the length of the trace (`PP_GetTraceLength`)
- Allocate an array equal to or larger than trace length
- Get a trace (PP_GetTrace)
- Move the current trace to the analysis trace (PP_`CurrTrace2AnalysisTrace`)
- Get the peaks orders by index (`PP_GetPeaks_Idx`)
- Using the first two peaks from the previous call get the edges of the first two pulses (`PP_GetPulseEdgesPosition`)
- Return edges of first two pulses

There are a number of approaches that will provide measurement results. You could also use the trace based measurements (e.g. `PP_GetTracePkPwr`) if they are sufficient.

Pass Parameters:

addr – address of the selected sensor
tr – a properly sized array of doubles
trLen – the length of the array allocated by the user
trUsed – the number of elements of the array containing valid data starting with the first element

Return Values:

Failure < 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetTrace(long addr, double *tr, long trLen, long* trUsed);
```

VB.NET

```
Public Declare Function PP_GetTrace Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef tr As Double, _
      ByVal trLen As Integer, _
      ByRef trUsed As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTrace
      (int addr,
      ref double tr,
      int trLen,
      ref int trUsed);
```

PP_GetTraceLength

Description: The trace varies with sweep time as shown in the table below. This call returns the number of trace points associated with the current sweep time.

Sweep Time	# Trace Points	Over Sampling	Resolution (time/points)
10 usec	480	96	0.02083 usec
20 usec	960	96	0.02083 usec
50 usec	2400	96	0.02083 usec
100 usec	4800	96	0.02083 usec
200 usec	9600	96	0.02083 usec
500 usec	10,000	48	0.05000 usec
1,000 usec	10,000	24	0.10000 usec
2,000 usec	10,000	24	0.20000 usec
5,000 usec	10,000	24	0.50000 usec
10,000 usec	10,000	24	1.00000 usec
20,000 usec	10,000	12	2.00000 usec
50,000 usec	10,000	6	5.00000 usec
100,000 usec	10,000	2	10.00000 usec
200,000 usec	10,000	1	20.00000 usec
500,000 usec	10,000	1	50.00000 usec
1,000,000 usec	10,000	1	100.00000 usec

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetTraceLength(long addr);
```

VB.NET

```
Public Declare Function PP_GetTraceLength Lib "LB_API2.dll" _
      (ByVal addr As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTraceLength(int addr);
```

PP_GetTraceAvgPower (and related calls)

PP_GetTraceAvgPower
PP_GetTraceCrestFactor
PP_GetTraceDC
PP_GetTracePkPwr
PP_GetTracePulsePower

Description: These calls make a number of measurements that are algorithmically similar to the measurements used in the power meter calls. However, these calls operate on a single trace (with may or may not be averaged) instead of a set of random samples. These measurement results may also differ in some cases with the gated measurements. Gate measurements require the user selects a particular cycle within a trace.

Normally these differences are small and unimportant and are more likely to be a distraction. However, there are times when these distinctions are important and account for differences in the measurements. It should be noted that these differences are not errors. Instead, the difference reflect the differences in how the data is acquired. To be precise the variations in results are a direct result of the differences how the data is selected.

Power meter measurements take a larger number of random samples over a specified period of time. This randomization tends to negate partial cycles (a potential issue with trace some based measurements) but this methodology may also include periods that the user regards as undesirable. While the measured result may be correct (give a specific set of sample), random samples may not always represent the best means of collecting the data for the users intended purpose.

The trace based measurements use contiguous sets of data in the form of a trace. These samples are time related to each other and related to certain features of the signal. Most notable among these features is the transition or edge.

In other words, trace based measurements selects data containing signal content directed by the user. Some of the elements that may affect the trace are trigger edge, trigger mode, pulse criteria, delay, trace averaging and averaging mode. The resultant acquisition may bias trace based measurements in an undesirable fashion. In this case the user should be aware of the potential for undesirable bias.

Gated measurements allow that the user to select and measure a specific portion of the signal. At the same time gated measurements deliberately ignore all other data. So that the critical element here is that the user select a representative subset of the visible trace. And the user should also be aware that potential exists for other signals to be present. It may be important to check for the presence of these signals.

In those cases where some additional assurance is desirable, you may want to consider using two methods. It is quite acceptable to use the power meter measurements along side gates measurements. Or you can also use these trace base measurements.

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_GetTraceAvgPower(long addr, double* avgPwr);
```

```
long __stdcall PP_GetTraceCrestFactor(long addr, double* CrF);
long __stdcall PP_GetTraceDC(long addr, double* dutyCycle);
long __stdcall PP_GetTracePkPwr(long addr, double* pkPwr);
long __stdcall PP_GetTracePulsePower(long addr, double* plsPwr);
```

VB.NET

```
Public Declare Function PP_GetTraceAvgPower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef avgPwr As Double) As Integer
Public Declare Function PP_GetTraceCrestFactor Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef CrF As Double) As Integer
Public Declare Function PP_GetTraceDC Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef dutyCycle As Double) As Integer
Public Declare Function PP_GetTracePkPwr Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef pkPwr As Double) As Integer
Public Declare Function PP_GetTracePulsePower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef plsPwr As Double) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTraceAvgPower
      (int addr,
      ref double avgPwr);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTracePulsePower
      (int addr,
      ref double plsPwr);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTraceCrestFactor
      (int addr,
      ref double CrF);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTracePkPwr
      (int addr,
      ref double pkPwr);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTraceDC
      (int addr,
      ref double dutyCycle);
```

PP_SetTriggerEdge (and related calls)

PP_SetTriggerEdge
PP_GetTriggerEdge

Description: Sets the trigger signal edge on which the beginning of the trace will occur. The values are positive edge or negative edge.

Pass Parameters:
addr – address of the selected sensor
TrgEdge – specifies the trigger edge. This value can be 0 (positive) or 1 (negative)

Return Values:
Failure <= 0
Success >= 1

Declarations:

C++

```
enum TRIGGER_EDGE
{
	POSITIVE = 0,
	NEGATIVE = 1
};

long __stdcall PP_SetTriggerEdge(long addr, TRIGGER_EDGE TrgEdge);
long __stdcall PP_GetTriggerEdge(long addr, TRIGGER_EDGE* TrgEdge);
```

VB.NET

```
Public Enum TRIGGER_EDGE
	POSITIVE = 0
	NEGATIVE = 1
End Enum

Public Declare Function PP_SetTriggerEdge Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByVal trgEdge As TRIGGER_EDGE) As Integer
Public Declare Function PP_GetTriggerEdge Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByRefl trgEdge As TRIGGER_EDGE) As Integer
```

C Sharp

```
public enum TRIGGER_EDGE
{
	POSITIVE = 0,
	NEGATIVE = 1
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTriggerEdge(int addr, TRIGGER_EDGE TrgEdge);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTriggerEdge(int addr, ref TRIGGER_EDGE TrgEdge);
```

PP_SetTriggerLevel (and related calls)

PP_SetTriggerLevel
PP_GetTriggerLevel

Description: Sets the trigger level for internal triggering (manual or automatic). The level is specified in dBm. How this value is used depends to some extent on trigger edge and threshold. If the edge is positive the trace will be triggered by the first sample whose value equals or exceeds the trigger level. If the edge is negative the trace will be triggered by the first sample whose value is equal to or less than the trigger level.

Pass Parameters:

addr – address of the selected sensor

TrgLvl – the trigger level value in dBm.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetTriggerLevel(long addr, double TrgLvl);
long __stdcall PP_GetTriggerLevel(long addr, double* TrgLvl);
```

VB.NET

```
Public Declare Function PP_SetTriggerLevel Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal trgLvl As Double) As Integer
Public Declare Function PP_GetTriggerLevel Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef trgLvl As Double) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTriggerLevel
      (int addr,
      double TrgLvl);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTriggerLevel
      (int addr,
      ref double TrgLvl);
```

PP_SetTriggerOut (and related calls)

PP_SetTriggerOut
PP_GetTriggerOut

Description: Sets or gets the trigger out mode. The trigger out can be off (no trigger out) or it can be normal (same polarity as the input trigger or inverted relative to the input trigger.

Pass Parameters:

addr – address of the selected sensor

trgOutMode – sets or gets the mode.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum TRIGGER_OUT_MODE
{
	TRG_OUT_DISABLED = 0,
	TRG_OUT_ENABLED_NON_INV = 1,
	TRG_OUT_ENABLED_INV = 2
};

long __stdcall PP_SetTriggerOut(long addr, TRIGGER_OUT_MODE trgOutMode);
long __stdcall PP_GetTriggerOut(long addr, TRIGGER_OUT_MODE *trgOutMode);
```

VB.NET

```
'TRIGGER_OUT_MOD 11/19/2008
Public Enum TRIGGER_OUT_MODE
	TRG_OUT_DISABLED
	TRG_OUT_ENABLED_NON_INV
	TRG_OUT_ENABLED_INV
End Enum

Public Declare Function PP_SetTriggerOut Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByVal trgOutMode As TRIGGER_OUT_MODE) As Integer
Public Declare Function PP_GetTriggerOut Lib "LB_API2.dll" _
	(ByVal addr As Integer, _
	ByRef trgOutMode As TRIGGER_OUT_MODE) As Integer
```

C Sharp

```
public enum TRIGGER_OUT_MODE
{
	TRG_OUT_DISABLED = 0,
	TRG_OUT_ENABLED_NON_INV = 1,
	TRG_OUT_ENABLED_INV = 2
}
```

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTriggerOut
      (int addr,
      TRIGGER_OUT_MODE trgOutMode);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTriggerOut
      (int addr,
      ref TRIGGER_OUT_MODE trgOutMode);
```

PP_SetTriggerSoure (and related calls)

PP_SetTriggerSoure
PP_GetTriggerSoure

Description: Trigger source can be internal or external. External triggers are received via the SMB connector on the back of the sensor. External triggers are TTL triggers. The external trigger must have the following characteristics:

- pulse width of at least 2 usec
- PRF <= 300kHz

Internal triggers are derived from the incoming signal (most like an o'scope internal triggering). If the source is internal auto level the following algorithm is followed:

- take a single untriggered sweep
- examine the single sweep for a peaks and transitions
- set the trigger level to the peak – peak criteria (typically 3-6dB)
- take a normal trace triggering on the previously selected value

This process is followed each time a trace is taken. If the source is set to internal manual the incoming trace is examined for an appropriate negative or positive edge at the level specified in PP_SetTriggerLevel. If a signal is not found an error is returned.

Pass Parameters:

addr – address of the selected sensor

TrgSrc – the trigger source, internal auto-level, internal manual level and external.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
enum TRIGGER_SOURCE
{
	INT_AUTO_LEVEL = 0,
	INTERNAL = 1,
	EXTERNAL = 2
};

long __stdcall PP_SetTriggerSoure(long addr, TRIGGER_SOURCE TrgSrc);
long __stdcall PP_GetTriggerSoure(long addr, TRIGGER_SOURCE* TrgSrc);
```

VB.NET

```
Public Enum TRIGGER_SOURCE
	INT_AUTO_LEVEL = 0
	INTERNAL = 1
	EXTERNAL = 2
End Enum
```

```
Public Declare Function PP_SetTriggerSoure Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal trgSrc As TRIGGER_SOURCE) As Integer
Public Declare Function PP_GetTriggerSoure Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByRef trgSrc As TRIGGER_SOURCE) As Integer
```

C Sharp

```
public enum TRIGGER_SOURCE
{
      INT_AUTO_LEVEL = 0,
      INTERNAL = 1,
      EXTERNAL = 2
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTriggerSoure
      (int addr,
      TRIGGER_SOURCE TrgSrc);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTriggerSoure
      (int addr,
      ref TRIGGER_SOURCE TrgSrc);
```

PP_MarkerToPk (and related calls)

PP_MarkerToPk
PP_MarkerToLowestPk
PP_MarkerToFirstPk
PP_MarkerToLastPk
PP_MarkerPrevPk
PP_MarkerNextPk
PP_MarkerPkHigher
PP_MarkerPkLower

Description: Sets one of five markers (0<=mrkIdx<=4) to the position specified in the call. The underlying algorithm for all of the calls begins by getting a list of the peaks ordered by index or value as is deemed most appropriate. The subsequent action associated with the various calls are as follows.

- Marker to peak sets the marker to the highest peak
- Marker to lowest peak sets the marker to the lowest peak
- Marker to first peak sets the marker to the left most peak
- Marker to last peak sets the marker to the right most peak
- Marker to previous peak sets the marker to the peak to the left of the current location
- Marker to next peak sets the marker to the peak to the right of the current location
- Marker to next higher peak sets the marker to the first peak greater than the current value.
- Marker to next lower peak sets the marker to the first peak less than the current value.

Note that the mode of the selected marker must be normal or delta. Otherwise an error will be returned. If the mode is normal then the normal marker is repositioned. If the mode is delta then the delta marker is repositioned.

Pass Parameters:

addr – address of the selected sensor
mrkIdx – index of the marker (0..4)

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_MarkerToPk(long addr, long mrkIdx);
long __stdcall PP_MarkerToLowestPk(long addr, long mrkIdx);
long __stdcall PP_MarkerToFirstPk(long addr, long mrkIdx);
long __stdcall PP_MarkerToLastPk(long addr, long mrkIdx);
long __stdcall PP_MarkerPrevPk(long addr, long mrkIdx);
long __stdcall PP_MarkerNextPk(long addr, long mrkIdx);
long __stdcall PP_MarkerPkHigher(long addr, long mrkIdx);
long __stdcall PP_MarkerPkLower(long addr, long mrkIdx);
```

VB.NET

```
Public Declare Function PP_MarkerToPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
```

```
      As Integer
Public Declare Function PP_MarkerToLowestPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerToFirstPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerToLastPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerPrevPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerNextPk Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerPkHigher Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
Public Declare Function PP_MarkerPkLower Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerToPk(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerToLowestPk(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerPkLower(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerPkHigher(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerToFirstPk(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerToLastPk(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerPrevPk(int addr, int mrkIdx);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerNextPk(int addr, int mrkIdx);
```

PP_MarkerPosIsValid

Description: Returns the state of the selected marker. The marker mode must be normal or delta first. Otherwise an error will be returned. For the trace index of the marker position must be equal to or greater than zero (the beginning of the trace) and less than the trace length (end of the trace). See the table located in the PP_SetSweepTime description for more information about trace length.

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker (0..4)

valid – return value is 0 if the marker position is invalid and 1 if it is valid.

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_MarkerPosIsValid(long addr, long mrkIdx, long* valid);
```

VB.NET

```
Public Declare Function PP_MarkerPosIsValid Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal mkrIdx As Integer, _
      ByRef valid As Integer) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_MarkerPosIsValid
      (int addr,
      int mrkIdx,
      ref int  valid);
```

PP_SetAnalysisTrace

Description: Allows user to place a previously acquired trace into the analysis buffer. The allows a user to acquire a substantial set of data at one point and time and then perform the analysis (using gates, trace based measurements and markers) at some later time.

Pass Parameters:

addr – address of the selected sensor
frequency – frequency at which the trace was acquired
sweeptime – sweep time of the trace (in microseconds)
*tr – pointer to the first array element of a trace
trLen – length of the trace array
units – power units (should be dBm)

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetAnalysisTrace(long addr,
      double frequency,
      double sweepTime,
      double*tr,
      long trLen,
      PWR_UNITS units);
```

VB.NET

```
Public Declare Function PP_SetAnalysisTrace Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal frequency As Double, _
      ByVal sweepTime As Double, _
      ByRef tr As Double, _
      ByVal trLen As Integer, _
      ByVal units As PWR_UNITS) _
      As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetAnalysisTrace
      (int addr,
      ref double tr,
      int trLen,
      int units);
```

PP_SetClosestSweepTimeUSEC

Description: Sets the sweep time to the fixed sweep time closest to the sweep time sent (in microseconds) to the routine. For instance, if a value of 11 was sent (meaning 11 usec sweep time) the system would set the sweep time to 10 usec.

Pass Parameters:

addr – address of the selected sensor
swpTm – desired sweep time in microseconds

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetClosestSweepTimeUSEC(long  addr, long  swpTm);
```

VB.NET

```
Public Declare Function PP_SetClosestSweepTimeUSEC Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal swpTimeUSEC As Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetClosestSweepTimeUSEC(long  addr, long  swpTm);
```

PP_SetSweepHoldOff

Description: Sets the minimum period between the end of one trace and the beginning of the next trace. During this period potential trigger events are ignored. Sweep hold should be used cautiously with faster sweep times (where under sampling is more common). For instance, if a 1 msec (or 1000 usec) sweep time is being used, then 24x under sampling is being used. And since each under sampled trace must be synchronized, the user will experience one hold off period between each acquisition…or in this case 23 hold off periods. So that if the hold off was set to 10,000 usec (10msec) the user would experience an additional 230 msec of data acquisition time due to hold off.

Pass Parameters:

addr – address of the selected sensor
SwpHOff – sweep time hold off in microseconds

Return Values:

Failure <= 0
Success >= 1

Declarations:

C++

```
long __stdcall PP_SetSweepHoldOff(long addr, long SwpHOff);
```

VB.NET

```
Public Declare Function PP_SetSweepHoldOff Lib "LB_API2.dll" _
      (ByVal addr As Integer, _
      ByVal SwpHOff as Integer) As Integer
```

C Sharp

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetSweepHoldOff(int addr, int SwpHOff);
```