Programming Guide for the

PULSE PROFILE APPLICATION

MODEL LB480A

TABLE OF CONTENTS

NOTICES

© LadyBug Technologies LLC 2007

This document contains information which is copyright protected. Do not duplicate without permission or as allowed by copyright laws.

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 LB480A Programming Guide for the Pulse Profile Application).

CONTACT INFORMATION

LadyBug Technologies LLC
3345 Industrial Drive, Suite 10
Santa Rosa, CA 95403
Phone 707.546.1050
Fax 707.237.6724
www.ladybug-tech.com

Introduction

This document is a preliminary programming guide for the LB480A model power sensor using the ***Pulse Profile Application****.* Note that all sensor models can run the ***Power Meter Application***. At this time, however, only the LB480A is capable of making pulse profile measurements. This document details functions listed to the ***Pulse Profiling Application***, but also includes some general information functions.

Note that all functions are prefixed with either an *"LB"* or *"PP"*. *"LB"* functions apply to both general purpose and power meter calls whereas *"PP"* functions apply exclusively to pulse profiling calls. For example, setting or retrieving the sweep time is clearly a pulse profiling call and not needed for a power meter application. Therefore, the function names are *PP_SetSweepTime* and *PP_GetSweepTime*.

The general purpose calls that apply to both pulse profile and power meter applications are listed at the beginning of the reference section of this document. This includes such calls as *LB_Initialize_Addr* (used to initialize a sensor) and *LB_BlinkLED_Addr* (used to identify a sensor physically). Complete documentation on all *"LB"* general purpose and power meter functions can be found in the Power Meter Programming Guide.

To provide access to the most common development environments, the programmatic interface for all *"LB"* and *"PP"* functions consists of a dynamic link library or DLL. This library uses the WinAPI or "_stdcall" calling convention. It is located in the Ladybug application directory, with the default location being *"C:\Program Files\Ladybug Technologies LLC\Ladybug Pulse Profiling Application\ LB_API2.DLL".*

Included in the product installation are various demonstration programs written in VB 6.0, VB.NET, LabVIEW 8.5 and VEE 7. Almost all DLL functions are demonstrated in these examples. One such application is *TestHarness2*, which can be found in a sub-directory of the same name in the application directory. A header file and lib file are also included *(LB_API2.h and LB_API2.lib).*

The DLL is compiled using Visual Studio 2005 Visual C++. 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.

LabVIEW users should note that many (but not all) of these functions have LabVIEW equivalents. The location of these VI's are listed in the function descriptions. Some VI's, however, have different inputs and/or different units. For example, all of the time functions in the DLL use microseconds whereas the LabVIEW functions use seconds. Refer to the on-line LabVIEW help for each VI for more information.

Making a Simple Measurement

The purpose of this section is to get the user 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 a 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 before proceeding.

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 which is denoted by an index of 1.

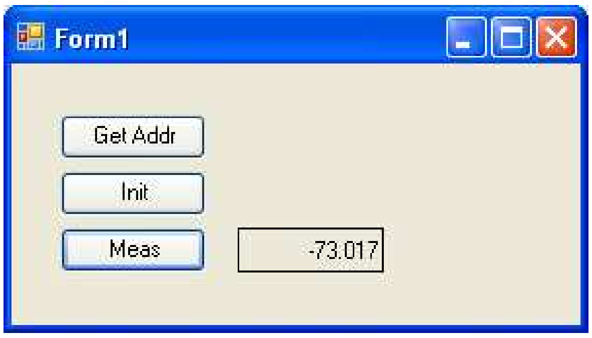
Using the address from the first call, we can initialize the sensor by clicking the second button on the form. This makes the call “LB_InitializeSensor_Addr”, which can be interpreted as “initialize the sensor using the address”. Initialization causes the calibration constants and other information for 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.

To perform a CW measurement using "LB_CWMeasure”, click on the third button. 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) and 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 perform measurements as often as we like. We can also change sensor states and perform measurements.

Using the Application:

After compiling, the application should appear as follows when running:



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 (repeat as desired). A measurement should appear in the label. Once the instrument has been initialized, the button can be clicked repeatedly.

A few items of interest to some programmers:

- "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 and 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 possible to pass only simple structures this way via a _stdcall. This does not work for more complex structures in our experience.

NOTE: Code modifications may be required for earlier versions of Visual Studio.NET.

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 was done primarily through GPIB and required the use of addresses. This approach enabled the development of flexible software. GPIB addresses were typically set at the front panel of the instrument or by using switches on the back of the instrument (and sometimes inside the instrument).

As a cursory inspection will show, Ladybug sensors do not have switches or front panels. So, how does the PC control or communicate with a Ladybug sensor? The following questions become important:

- How do I determine 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 to the same PC?
- 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 and code available on the product CD to set and check instrument addresses. Feel free to examine these applications to reinforce this explanation and aid in development.

The first step in instrument communication is to identify the instrument. Note that the serial number can be found on the back of each sensor and that there is also a green LED (power light). There are calls to perform the following functions:

- Collect all sensor identification information (index, serial number and address)
- Obtain the address by serial number or index
- Set or change the address by using the index, serial number, or current address
- Retrieve the serial number using the index or address
- Retrieve the index using the serial number or address
- Blink the LED on a specific sensor
- Determine if an address conflict exists
- Determine if changing an address will cause an address conflict

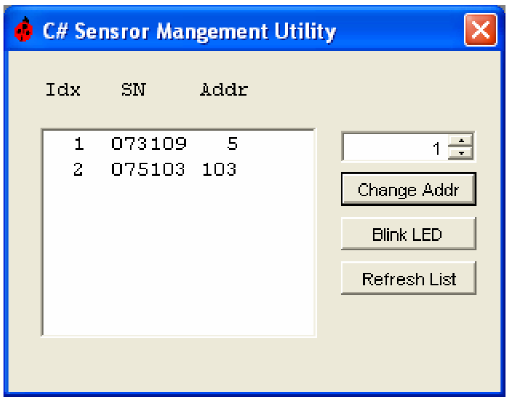
Each of these functions are described in the following sections in a straightforward manner to facilitate your programming task.

Although some of the API functions use the serial number or index, most use the address exclusively. The functions that do use the serial number or index tend to be management functions. For more details, see the declarations in the sample VB 6.0, VB.NET and C SHARP projects.

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.

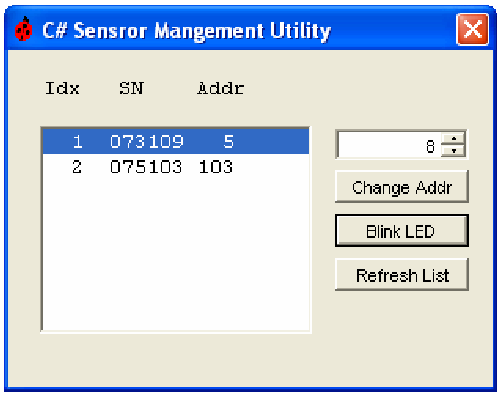
The application should display a list of sensor(s) currently attached to your computer. Each sensor is represented by an index; a serial number (stamped on the back of the sensor); and an address.



```
Idx    SN      Addr

  1   073109     5
  2   075103   103
```

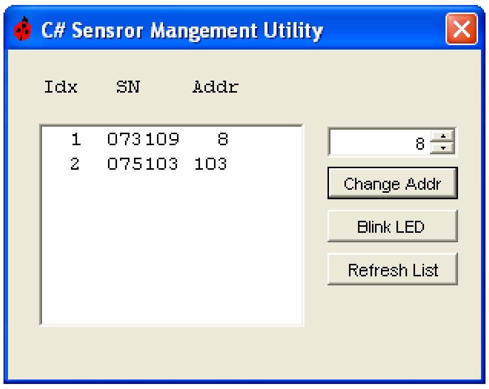
Select a sensor as shown below. Use the up/down arrows to set the desired address in the numeric control on the right. In the picture below, we have chosen the sensor with a serial number of “073109” and will change the address from 5 to 8. You can use the “Blink LED” button to ensure that you are addressing the correct sensor. The sensor’s LED should blink four times in quick succession.



```
Idx    SN      Addr

  1   073109     5
  2   075103   103
```

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.



```
Idx    SN      Addr

  1   073109    8
  2   075103  103
```

Select the sensor in the list box and click “Blink LED” to identify the sensor whose address was just changed.

Close the application once you have the sensor set to the address of choice. *The address is set in non-volatile memory and will retain its value after power is removed*.

NOTE: This can also be performed programmatically 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)

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 can assign any legitimate address (1-255) to any sensor.

The address is stored in non-volatile memory and will not be 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, 0 otherwise

Sensor indices are assigned arbitrarily by order of identification beginning with 1 and continuing in consecutive order. Typically, the index is less useful than the address and serial number although it is provided for completeness . It 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 example, 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 of a sensor 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 - retrieve 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 – retrieve a number indicating 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 – retrieves the cal due date

Finally, we can discuss the address. Almost all other calls - getting, setting measurement attributes and making measurements - require the address. There are over 80 such functions. A few of the more commonly used of these 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 – determines if a sensor with the given address is connected to the system
  - LB_GetSerNo_Addr – retrieves the serial number of the sensor at the address
  - LB_InitializeSensor_Addr – initializes the sensor
  - LB_BlinkLED_Addr – blinks the LED (useful in identifying the units physical location)
- 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 – retrieves the frequency (Hz)
  - LB_SetAverages – sets the number of averages
  - LB_GetAverages – retrieves the number of averages
  - LB_SetMeasurementPowerUnits – sets the measurement units to dBm, dBW, dBkW, dBuV, V or W
  - LB_GetMeasurementPowerUnits – retrieves the measurement units

See the power meter programming guide and the rest of this document for details on all of the functions.

Function Calls for The Pulse Profiling Application – Model LB480A ONLY

All of the functions detailed in this document are exported from a Visual C++ 2005 project using a _stdcall calling convention. The declarations are also available for C SHARP, VB 6.0, and VB.NET in the example code.

LIBRARY	"LB_API2"

Please contact us if a calling convention other than _stdcall is required. We may be able to supply it on a case by case basis.

Refer to the READ.ME file for information on how to complete a development environment installation. For LabVIEW users, the LabVIEW installation should be adequate.

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 important 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. A return value of 0 or less indicates an invalid trace. Note that all measurements, gates and marker functions operate on the analysis trace and not the measurement trace. An analysis trace is obtained most commonly by calling `PP_CurrTrace2AnalysisTrace` after having taken a measurement (see `PP_GetTrace`).

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data → Ana Trace Valid

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.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Success >= 1

PP_CnvtTrace

Description: Converts a trace (trIn) to different units 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.

Prototype:

```
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
};

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

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

PP_CurrTrace2AnalysisTrace

Description: The driver can hold two traces for each initialized sensor, the current trace and the analysis trace. The current trace is the most recent measurement. The analysis trace is the trace data used to make calculations. This call copies the current trace to the analysis trace and returns a copy of that trace.

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data → Curr to Ana

PP_GatePositionIsValid

Description: Determines if the specified gate is valid. The gate index may be between 0 and 4. For the gate to be valid, the following conditions must be true:

- 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
- The left side of the gate must precede the right side of the gate and they must be separated by at least two data points

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Gate Pos Valid?

PP_SetAvgMode
PP_GetAvgMode
PP_SetTraceAvgs
PP_GetTraceAvgs
PP_ResetTraceAveraging

Description: Trace averaging can be important in making good measurements. In any case, averaging will reduce the noise on the trace. There are three elements to trace averaging, the mode, the number of traces to average, and 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, averaging will not be performed. If it is auto reset, averaging will restart (all previous averages will be discarded) when the auto reset criteria is satisfied. If averaging mode is manual reset, averaging will continue until a call is made to either change the averages, turn the averaging off, or reset the averaging.

PP_Set(Get)TraceAvgs sets (or gets) the number of traces to be averaged This number must be between 1 and 100.

PP_ResetTraceAveraging restarts the averaging process with the next trace if the mode is auto reset or manual reset.

Prototypes:

```
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);
```

Pass Parameters:

addr – address of the selected sensor

*mode – pointer to AVG_MODE (32 bit integer)

averages – number of trace averages

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Averaging

PP_SetAvgResetSens
PP_GetAvgResetSens

Description: This sets (or gets) the criteria for resetting trace averaging. It applies only when the averaging mode is `AVG_AUTO_RESET` (see PP_GetAvgMode and PP_SetAvgMode).

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Averaging

PP_SetFilter
PP_GetFilter
PP_SetPoles
PP_GetPoles

Description: Sets (or gets) the enum value of the filter. Both the frequency of the filter and the number of poles can be set or retrieved. The poles vary the slope of the filter skirt while the cutoff varies the 3dB frequency of the filter.

Note that to be able to vary the filter setting above 100kHz the sensor must have option 004 installed (filter options).

The enums for the various filter poles and corner frequencies are shown below.

Prototype:

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

Pass Parameters:

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

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Filter

PP_GetGateAveragePower

Description: Returns the average power of the span in the analysis trace specified by the gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

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

Return Values:

Failure <= 0
Success >= 1

Declarations:

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Average Power

PP_GetGateCrestFactor

Description : Returns the crest factor (in dB) of the span in the analysis trace specified by the gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). Crest factor is always returned in dB and is equal to the peak power – average power between the gate edges.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*crFactor – returns the crest factor in dB

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Crest Factor

PP_GetGateDroop

Description : Returns the droop of the span in the analysis trace specified by the gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). 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.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*droop – returns the droop of the signal in dB.

Return Values:

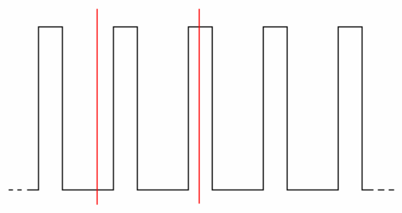
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Droop

PP_GetGateDutyCycle

Description: Returns the duty cycle (as a decimal) of the span in the analysis trace specified by the gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). The gate may contain many pulses. It must, however, 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, only the first full cycle will be used to make the measurement. The value returned is a decimal value between 0 and 1. 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

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*dutyCycle – returns the ratio of on time to off time

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Duty Cycle

PP_GetGateEndPosition

Description: Returns the location of the right side of the specified gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). The location returned is the index of the analysis trace of the right or ending side of the gate. Note that the analysis trace is a zero based array. Refer to PP_CurrTrace2AnalysisTrace for more details of the analysis trace.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*trIdx – returns the trace index (see PP_CurrTrace2AnalysisTrace).

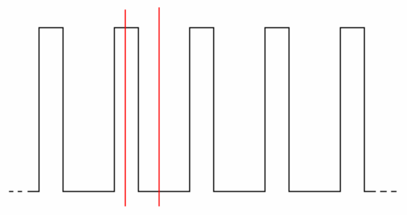
Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get End Pos

PP_GetGateFallTime

Description: Returns the fall time, in microseconds, of the pulse contained in the selected gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). The gate must be properly positioned to return a meaningful value. The left side of the gate must be positioned between a pulse rising and falling edge and the right side must be positioned after the next falling edge. 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.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*fallTm – returns the fall time, in microseconds

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Fall Time

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.

Prototype:

```
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);
```

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*mode – the gate mode

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate

PP_GetGateOverShoot

Description: Returns the overshoot in dB. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

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 quarter and the average of the last three quarters

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*overShoot –overshoot in dB

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Overshoot

PP_GetGatePeakPower

Description: Returns the maximum power of the analysis trace in the selected gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*pkPwr – returns the peak power

Return Values:

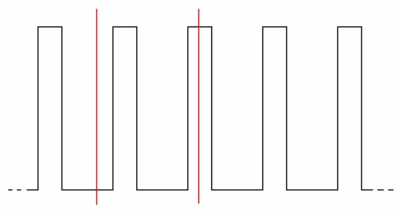
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Peak Power

PP_GetGatePRF

Description: Returns the pulse repetition frequency (PRF) in Hertz for the specified gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

PRF is defined as the reciprocal of the time between consecutive rising edges. If a complete pulse followed by a rising edge is not present in the span defined by the gate an error is returned. Note that a complete pulse is a rising edge followed by falling edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate with gate edges shown in red. It is acceptable for the gate to contain many pulses. Only the first two rising edges, however, will be used to make the calculation.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*PRFreg – returns the frequency in Hertz.

Return Values:

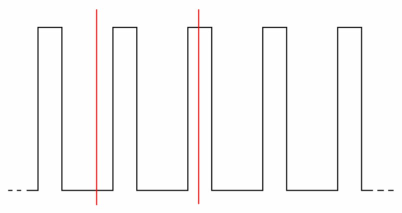
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get PRF

PP_GetGatePRT

Description: Returns the pulse repetition time (PRT) in microseconds using the same algorithm defined for the PP_GetGatePRF function. The sole difference is that time instead of frequency is returned. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

PRT is defined as the time between consecutive rising edges. If a complete pulse followed by a rising edge is not present in the span defined by the gate an error is returned. Note that a complete pulse is a rising edge followed by falling edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate with gate edges shown in red. It is acceptable for the gate to contain many pulses. Only the first two rising edges, however, will be used to make the calculation.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*PRTime – returns the time in microseconds.

Return Values:

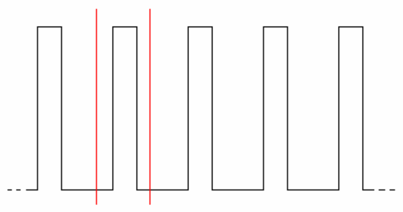
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get PRT

PP_GetGatePulseWidth

Description: Measures the pulse width in microseconds for the specified gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

The span defined by the gate must contain at least one complete pulse. Note that a complete pulse is a rising edge followed by 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. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. Only the first complete pulse, however, will be used to make the measurement.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*plsWidth – returns pulse width in microseconds.

Return Values:

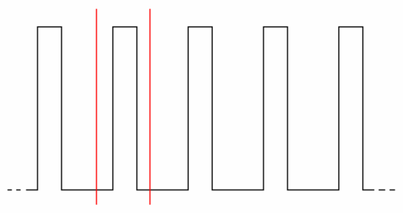
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Pulse Width

PP_GetGatePulsePower

Description: Returns the average pulse power for the specified gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function).

The span defined by the gate must contain at least one complete pulse. Note that a complete pulse is a rising edge followed by falling edge. The average pulse power is measured by averaging all of the samples between the rising edge and the subsequent falling edge. The diagram below depicts the minimum acceptable span defined by the edges of the gate. Gate edges are shown in red. It is acceptable for the gate to contain many pulses. . Only the first complete pulse, however, will be used to make the measurement.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*plsPwr – returns pulse power in dBm.

Return Values:

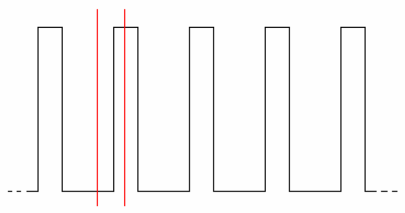
Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Pulse Power

PP_GetGateRiseTime

Description: Returns rise time in microseconds.

Description: Returns the rise time, in microseconds, of the pulse contained in the selected gate. Note that the gate must be valid (as described in the PP_GatePositionIsValid function) and the gate must be on (as described in the PP_GetGateMode function). The gate must be properly positioned to return a meaningful value. The left side of the gate must be positioned between the falling and rising edges of a pulse and the right side must be positioned after the next rising edge. 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.



Prototype:

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

Pass Parameters:

addr – address of the selected sensor

gateIdx – index of the selected gate

*riseTm – Measured rise time in microseconds

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate → Get Rise Time

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) position(s) in terms of trace index or time. If the index or time is 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 and an error will be returned. An error will also be returned if the gate start is positioned after the gate end.

All times are in microseconds and all indices are integers. Note that traces are zero-based arrays.

Prototype:

```
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 stpTm);
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_SetGateEndPosition(long addr, long gateIdx, long trIdx);
long __stdcall PP_GetGateEndPosition(long addr,long gateIdx,long* trEndIdx);
long __stdcall PP_SetGateStartTime(long addr,long gateIdx,double sttTm);
long __stdcall PP_GetGateStartTime(long addr,long gateIdx,double* sttTm);
long __stdcall PP_SetGateEndTime(long addr, long gateIdx, double gateTm);
long __stdcall PP_GetGateEndTime (long addr,long gateIdx,double* stpTm);
```

Pass Parameters:

addr – address of the selected sensor

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

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

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

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Gate

PP_GetMarkerAmp

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

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker → Get Amp

PP_GetMarkerDeltaAmp

Description: Returns the difference in amplitude between the normal marker and the delta marker in dBm. Note that the specified marker must be set to delta mode or an error will be returned.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker (0..4)

`*deltaMkrAmp` – amplitude (in dBm) of the trace at the normal marker minus the amplitude of the trace at the delta marker

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker → Get Delta Amp

PP_SetMarkerDeltaTime
PP_GetMarkerDeltaTime

Description: Sets (or gets) the delta position of the selected marker in microseconds. Note that an error will be returned if the position of the delta marker is less than 0 or greater than the sweep time.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

mrkIdx – index of marker (0..4)

mrkTm – time in microseconds

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker

PP_SetMarkerMode
PP_GetMarkerMode

Description: Sets (or gets) the marker mode. Marker mode can be either Off, Normal, or Delta. In normal mode, the normal marker is positioned or measured. In delta mode, the delta marker is positioned or measured.

Prototype:

```
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);
```

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.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker

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, the normal marker is positioned or returned. If the marker is in delta mode then the delta marker is positioned or returned and the normal marker is unaffected. The marker may be positioned in terms of index or time (microseconds).

Prototype:

```
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);
```

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker

PP_SetMeasurementThreshold
PP_GetMeasurementThreshold

Description: Sets (or gets) the measurement threshold. The measurement threshold and the peak criteria affect a number of measurement routines, most notably the peak routines. In short, the threshold is the lowest value considered in the trace. Before searching the analysis trace for peaks, all trace values lower than the threshold are set equal to the threshold. The threshold is set or reported in dBm.

In general, the threshold should be regarded as a filter.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

measThreshold_dBm – measurement threshold in dBm

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Peak

PP_GetPeaks_Val
PP_GetPeaks_Idx
PP_GetPeaksFromTr_Val
PP_GetPeaksFromTr_Idx
PP_GetPeaks_Val_Flat
PP_GetPeaks_Idx_Flat
PP_GetPeaksFromTr_ Val_Flat
PP_GetPeaksFromTr_Idx_Flat

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

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 the largest number of peaks. Typically, a safe array size is half the length of the trace (see the PP_SetSweepTime) because a rise and fall is required to identify a peak. This means that at a minimum of two points is required for each peak.

Depending on your development environment you may choose to use the _Flat calls. These calls do not use the Peak structure. Instead, they flatten the array of structures into two arrays of longs and doubles. This is used in environments such as Agilent VEE and National Instruments LabVIEW. Other environments may also find these calls more suitable.

Prototype:

```
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_Idx_Flat (long addr,
                              long* pkIndicies,
                              double* pkValues,
                              long maxPks,
                              long* pksUsed);
long __stdcall PP_GetPeaks_Val_Flat (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);
long __stdcall PP_GetPeaksFromTr_Idx_Flat(double* tr,
                              long trLen,
                              long units,
                              double peakCrit,
                              double measThresh,
                              long* trIdx,
                              double* value,
                              long maxPks,
                              long* pksUsed);
long __stdcall PP_GetPeaksFromTr_Val_Flat(double* tr,
                              long trLen,
                              long units,
                              double peakCrit,
                              double measThresh,
                              long* trIdx,
                              double* value,
                              long maxPks,
                              long* pksUsed);
```

Pass Parameters:

addr – address of the selected sensor

peaks – 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.

pkIndicies – an array of the peak positions (used for the flat functions)

pkValues – an array of the peak amplitudes (used for the flat functions)

tr – an array of data to be analyzed

trLen – the trace length, or number of samples in the trace

units – the units of measurement

peakCrit – peak criteria used to define a peak.

measThresh – measurement threshold use d to filter peaks.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Peak

PP_GetPulseEdgesTime
PP_GetPulseEdgesPosition

Description: Returns the index of the leading and trailing edges of the pulse containing the peak located at pkTime or pkIdx. These calls are intended to be used with PP_GetPeak and other routines as shown below in the algorithm for calculating rise time. It uses PP_GetPulseEdgesPosition but the same algorithm would work with PP_GetPulseEdgesTime also. The difference of course is that everything would be 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, which require at least two peaks. For this, evaluate the pksUsed parameter returned in the previous `PP_GetPeaks_Idx` call.
- Select the peaks of interest (pick the first peak returned in PP_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 edges

    Example:
    Assume a 1msec sweep time (10,000 points) for a resolution of 100 nsec
    Assume a 10kHz signal with a 20% duty cycle
    Assume the first peak is located at an index of 1100

    - The pulse is 200 points (or pixels) wide so that the pulse edges will be approximately:
      - Left pulse edge: 1000
      - Right pulse edge: 1200
    - Set the gate edges as follows:
      - Left side of the gate at 950 (about 50 pixels 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 related functions noted above are especially useful in making programmatic measurements by enabling easy placement of gate edges.

Prototype:

```
long __stdcall PP_GetPulseEdgesTime(long addr, double pkTime, double* leftSide,
double* rightSide)

long __stdcall PP_GetPulseEdgesPosition (long addr, long pkIdx, long *leftTrIdx, long
*rightTrIdx)
```

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Peak

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. Limitations are as follows:

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

Delay sweep is taken in one of two ways. Sweep times less than or equal to 10msec always use under sampling, which tends to extend the time required to acquire data. Traces taken without under sampling will show an improvement in data acquisition time but this may result in an increase in noise at lower power levels. Trace averaging can be used to decrease noise.

Prototype:

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

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

Pass Parameters:

addr – address of the selected sensor

SwpDly –delay in microseconds from the trigger edge until data is taken.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Sweep

PP_SetSweepDelayMode
PP_GetSweepDelayMode

Description: Sets (or gets) the sweep delay mode, either on or off. The sweep delay time is not changed by either of these calls. If the sweep delay mode is off, the trace will be acquired immediately after the trigger is detected. If it is on, the trace will be acquired after the sweep delay.

Prototype:

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

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

Pass Parameters:

addr – address of the selected sensor

SwpDlyMode – 0=OFF, 1 = ON

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Sweep

PP_SetSweepTime
PP_GetSweepTime

Description: Sets (or gets) the sweep time (in microseconds) of the next sweep taken by the sensor. 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	Over Sampling	Resolution (time/points)
10 usec	480	96	0.02833 usec
20 usec	960	96	0.02833 usec
50 usec	2400	96	0.02833 usec
100 usec	4800	96	0.02833 usec
200 usec	9600	96	0.02833 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

Prototype:

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

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

Pass Parameters:

addr – address of the selected sensor
SwpTm – sweep time, in microseconds

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configure → Sweep

PP_GetTrace

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

- Initialize the sensor (`LB_Initialize_Addr`)
- Set the frequency (`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 ordered 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 and 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 executed, a number of calls on subsequent measurements can be eliminated. The most notable is initialization. Calls such as setting frequency, sweep time, gate mode and others need not be made unless the state of the measurement changes. Assuming no changes, the following sequence would repeat the same measurement.

- 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 highly stable. Such approaches, however, 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:

- Initialize the sensor (`LB_Initialize_Addr`)
- Set the frequency (`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`)
- Use the current gate and trace to make a PRT measurement (`PP_GetGatePRT`)
- Use the current gate and trace to make a pulse width measurement (`PP_GetGatePulseWidth`)
- Use the same pulse edge information, reposition the gate edges to make 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 frequency (`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

The edges of the first two pulses can be used to make all of the measurements. Note that the trace based measurements (e.g. `PP_GetTracePkPwr`) may also be sufficient.

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data → Get Trace

PP_GetTraceLength

Description: Gets the length of the next sweep taken by the sensor. The number of points in 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	0.02833 usec
20 usec	960	96	0.02833 usec
50 usec	2400	96	0.02833 usec
100 usec	4800	96	0.02833 usec
200 usec	9600	96	0.02833 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

Prototype:

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

Pass Parameters:

addr – address of the selected sensor

Return Values:

Failure <= 0
Trace Length >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data → Get Length

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. These calls, however, operate on a single trace (which may or may not be averaged) instead of a set of random samples. These measurement results may also differ from the gated measurements. Gated measurements require the user to select a subset of the trace.

Normally, the differences between these three methods are small and unimportant and are more likely to be a distraction. This assumes that any gated measurements are setup properly. There are times, however, when these distinctions are important and account for differences in the measurements. It should be noted that these differences are not errors but reflect the differences in how the data is acquired. To be precise, the variations are a direct result of the differences in 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 some trace based measurements) but this methodology may also include periods that the user regards as undesirable. While the measured result may be correct (given 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 select 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 manner. In this case the user should be aware of the potential for undesirable bias.

Gated measurements allow the user to select and measure a specific portion of the signal and ignore all other data. The critical element is that the user properly selects a representative subset of the visible trace. 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, such as power meter measurements and gated measurements. You can also use these trace measurements.

The average power is calculated as the average power over the entire trace.

The crest factor is calculated as the peak power of the entire trace minus the average power of the entire trace.

The duty cycle is calculated as the ratio of “on” time to total time for the entire trace. Note that “on” time is the amount of time where the signal is above the peak power minus the peak criteria. For a peak power of 10 dBm and a peak criteria of 6 dBm, a data point would be considered “on” if it were above 4 dBm.

The peak power is the absolute maximum power level of the entire trace.

The pulse power is the average power of all of the data above the peak power minus the peak criteria (the “on” data).

Prototype:

```
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);
```

Pass Parameters:

addr – address of the selected sensor
avgPwr – average power
CrF – crest factor
dutyCycle – duty cycle
pkPwr – peak power
plsPwr – pulse power

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data

PP_SetTriggerEdge
PP_GetTriggerEdge

Description: Sets (or gets) the trigger edge for the selected sensor. The trigger edge may be either positive (0) or negative (1).

Prototype:

```
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);
```

Pass Parameters:

addr – address of the selected sensor

TrgEdge – the trigger edge.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configuration→Trigger

PP_SetTriggerLevel
PP_GetTriggerLevel

Description: Sets (or gets) the trigger level for internal triggering (manual or automatic). The level is specified in dBm. Note that this value is used differently depending on the trigger edge and threshold. If the edge is positive, the trace will be triggered by the first sample with a value greater than or equal to the trigger level. If the edge is negative, the trace will be triggered by the first sample with a value less than or equal to the trigger level.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor
TrgLvl – the trigger level value in dBm.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configuration→Trigger

PP_SetTriggerOut
PP_GetTriggerOut

Description: Sets (or gets) the trigger out mode. Trigger out can be any of the following:

1. Off (no trigger out)
2. Normal (same polarity as the input trigger)
3. Inverted relative to the input trigger.

Prototype:

```
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);
```

Pass Parameters:

addr – address of the selected sensor
trgOutMode – the trigger out mode.

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configuration→Trigger

PP_SetTriggerSoure
PP_GetTriggerSoure

Description: Sets (or gets) the trigger source. The trigger source must be internal (auto level), internal (manual), or external. External TTL triggers are received via the SMB connector on the back of the sensor. External triggers must have the following characteristics:

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

Internal triggers are derived from the incoming signal (similar to oscilloscope 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 minus the peak criteria (typically 3-6dB)
- Take a normal trace triggering on the previously selected value

This process is repeated 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 by PP_SetTriggerLevel. If a signal is not found an error is returned.

Prototype:

```
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);
```

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Configuration→Trigger

PP_MarkerToPk
PP_MarkerToLowestPk
PP_MarkerToFirstPk
PP_MarkerToLastPk
PP_MarkerPrevPk
PP_MarkerNextPk
PP_MarkerPkHigher
PP_MarkerPkLower

Description: Sets one of five markers (numbered 0 through 4) to the position specified in the call. The underlying algorithm for all of these calls begins by obtaining a list of the peaks. These must be ordered either by index or by value as is deemed most appropriate. The subsequent action associated with each of the 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 either normal or delta or an error will be returned. If the mode is normal, the normal marker is repositioned. If the mode is delta then the delta marker is repositioned.

Prototype:

```
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);
```

Pass Parameters:

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

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker

PP_MarkerPosIsValid

Description: Returns the state of the selected marker. The marker mode must be either normal or delta prior to calling this function. Otherwise an error will be returned. 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). Refer to the table located in the PP_SetSweepTime description for more information about trace length.

Prototype:

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

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

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Marker

PP_SetAnalysisTrace

Description: This function downloads a trace into the analysis trace on the sensor. Once downloaded, measurements such as duty cycle and average power may be made as if the analysis trace had been copied from the current trace. Note, however, that additional information is required, including the frequency of the measurement, the sweep time, the trace length, and the power units.

Prototype:

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

Pass Parameters:

addr – address of the selected sensor
frequency – frequency of the measurement, in Hertz
sweepTime – sweepTime, in microseconds
tr – the trace to be downloaded
trLen – the length of the trace, in samples
units – power units (refer to LB_ SetMeasurementPowerUnits for more details)

Return Values:

Failure <= 0
Success >= 1

LabVIEW Palette Location: Instrument I/O→ Instr Drivers →Pulse Power Sensor → Data

PP_SetClosestSweepTimeUSEC

Description: Sets the sweep time to a valid time nearest the desired value. All times are in microseconds. For example, if the desired time is 2200 microseconds, this function will set the sweep time to 2000 microseconds. Refer to the *PP_SetSweepTime* function for additional information on valid sweep times.

Prototype:

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

Pass Parameters:

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

Return Values:

Failure <= 0
Success >= 1