PSM3000, PSM4000, and PSM5000 Series
RF and Microwave Power Sensors/Meters

Programmer Manual

077-0651-00

Tektronix

PSM3000, PSM4000, and PSM5000 Series
RF and Microwave Power Sensors/Meters

Programmer Manual

www.tektronix.com

077-0651-00

Tektronix

Contacting Tektronix

Tektronix, Inc.
14150 SW Karl Braun Drive
P.O. Box 500
Beaverton, OR 97077
USA

For product information, sales, service, and technical support:

- In North America, call 1-800-833-9200.
- Worldwide, visit www.tektronix.com to find contacts in your area.

Table of Contents

Preface

This manual provides programming information for remotely controlling the following Tektronix products:

Product	Description
PSM3110	10MHz-8GHz (3.5mm-male)
PSM3120	10MHz-8GHz (N-Type male)
PSM3310	10MHz-18GHz (3.5mm-male)
PSM3320	10MHz-18GHz (N-Type male)
PSM3510	10MHz-26.5GHz (3.5mm-male)
PSM4110	10MHz-8GHz (3.5mm-male)
PSM4120	10MHz-8GHz (N-Type male)
PSM4320	50MHz-18GHz (N-Type male)
PSM4410	50MHz-20GHz (3.5mm-male)
PSM5110	100MHz-8GHz (3.5mm-male)
PSM5120	100MHz-8GHz (N-Type male)
PSM5320	50MHz-18GHz (N-Type male)
PSM5410	50MHz-20GHz (3.5mm-male)

This manual consists of the following sections:

- *Getting Started* describes how to make simple measurements using commands, provides command naming conventions, and describes how to communicate with the instrument. (See page 1, *Getting Started*.)
- *Command Groups* provides lists of commands by function and includes command descriptions. (See page 10, *Command Groups*.)
- *Commands Listed in Alphabetical Order* provides a list of commands that includes arguments, related commands, and programming examples. (See page 22, *Commands Listed in Alphabetical Order*.)
- *Index* provides an indexed list of the contents of this manual.

Getting Started

Most of the commands documented in this manual correspond directly to setup and measurement functions in the Power Meter and Pulse Profiling applications. Explanations of setup and measurement techniques are provided in the *PSM3000, PSM4000, and PSM5000 Series RF and Microwave Power Sensors/Meters User Manual*. Commands that correspond to the Power Meter application begin with the prefix "LB". Commands that correspond to the Pulse Profiling application begin with the prefix "PP".

The programmatic interface for these instruments consists of a dynamic link library (DLL). The name of the DLL is LB_API2.DLL. This library uses the WinAPI or "_stdcall" calling convention. The DLL is located in the Tektronix application directory. The name of the default application directory is "C:\Program Files\Tektronix\Tektronix Power Sensor Applications".

Included in the installation are the following files, which will help you become familiar with the available function calls. These files can be found in "C:\Program Files\Tektronix\Tektronix Power Sensor Applications\Sample Code":

- C# Pulse Profiling sample code
- C# Power Meter sample code
- VB Power Meter sample code
- C# High Speed Logger sample code
- Driver Installation Files

Three test harness programs are also included in the sample code installation. These executable programs are designed to exercise instrument functions and generate code that demonstrates correct syntax for commands. They can be found in subdirectories within the Sample Code directory. The different versions support different programming languages and instrument applications:

- C# Power Meter Applications
- C# Pulse Profiling Applications
- VB Power Meter Applications

NOTE. *See the README.txt file and test harness documentation in the Sample Code directory for more information about test harnesses.*

Commands Common to all Models

Some commands only apply to certain instrument models and their corresponding measurement capabilities. Other commands, called common commands, apply to all instrument models. These commands all begin with the prefix “LB”. This common command group includes commands to:

- Detect, identify, and address an instrument
- Initialize an instrument
- Manage communications and exceptions with an instrument
- Set the center frequency
- Perform average power measurements
- Configure and perform pass/fail limit testing on continuous wave (CW) power
- Set averaging parameters
- Set trigger conditions
- Configure offsets and relative measurements
- Save and recall setups

Pulse Measurements

The PSM4000 and PSM5000 Series instruments can measure power contained within pulses. These measurements include average pulse power, peak pulse power, crest factor, duty cycle, and average continuous wave (CW) power. Commands related to these measurement types begin with the prefix “LB”.

These instrument models support additional commands to:

- Set the criteria for distinguishing pulses
- Perform average pulse power, pulse power, crest factor, and duty cycle
- Configure and perform pass/fail limit testing on pulse power

Pulse Profiling

The PSM5000 Series instruments can perform pulse profiling measurements. Commands related to these measurement types begin with the prefix “PP”, and are used to:

- Configure and manage pulse profile triggers
- Perform gated measurements of pulse characteristics
- Manipulate markers and read back measurements
- Set filters
- Transfer the trace to the computer
- Perform power measurements on the trace
- Perform pulse measurements like rise/fall time, overshoot and droop

Addressing and Communicating with Sensors

You can use an instrument identifier and a series of function commands to establish a connection with the correct instrument. Function commands are provided for the following tasks:

- Collect all instrument identification information, including index, serial number, and address
- Obtain the address via serial number or index
- Set/change the address 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 instrument
- Determine if an address conflict exists
- Determine if changing an address will cause an address conflict

Identify the Instrument

You must first identify the instrument before you can use the function commands. You can do this in one of the following ways:

Address. A user-set identifier stored in the instrument's memory. The user has complete control over the address, and can assign any legitimate address (1-255) to any instrument.

Using the address is the recommended way to identify an instrument, because this eliminates the need to change the programming code if the original instrument is being replaced. More importantly, almost all commands require the instrument’s address, including getting, setting measurement attributes and making measurements (over 80 of them). The address is stored in non-volatile memory, so it is not lost when the instrument is disconnected or the system is powered down. Note that address conflicts may arise during the process of reassigning instrument addresses.

Serial Number. This number is permanent and determined by the factory. It is stamped into the back of the instrument.

The address or index can be retrieved using the serial number. You can also use the serial number to change the address and cause the LED to blink.

Index. A temporary logical descriptor determined by the system driver when the instrument is connected.

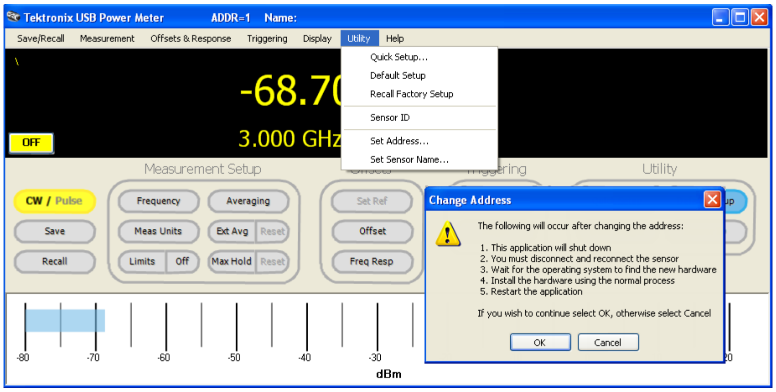
This is an arbitrary number that is assigned by order of identification. The index of the first instrument detected by the system is 1. The index of the second instrument is 2 and so on. Typically, the index is less useful than the address and serial number.

The index is most useful when coupled with the function call LB_SensorCnt. For instance, when LB_SensorCnt is called, if the instrument count is three, the first instrument discovered will have an index of 1; the second instrument will have an index of 2; and the third instrument will have an index of 3.

Set the Instrument Address

Next, you'll need to set the instrument address:

1. Open the Power Meter application, which should be visible in the Tektronix menu (Start > Tektronix > Power Meter Application > Power Meter Application.
2. After the Power Meter application opens, select the "Set Address" command found under the Utility menu tab. A window will open that lists the steps to follow in order to change the instrument address as shown below.

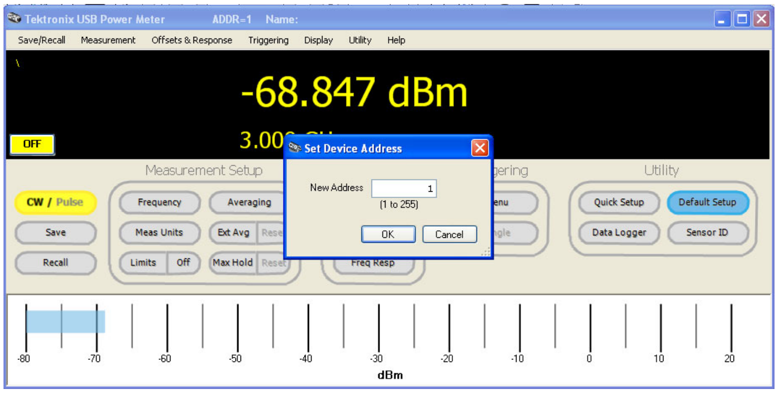


The following will occur after changing the address:

1. This application will shut down
2. You must disconnect and reconnect the sensor
3. Wait for the operating system to find the new hardware
4. Install the hardware using the normal process
5. Restart the application

If you wish to continue select OK, otherwise select Cancel

3. After pressing "Ok", set the new address of the device and press "Ok", as shown below. Then allow the application to shut down.



New Address 1
(1 to 255)

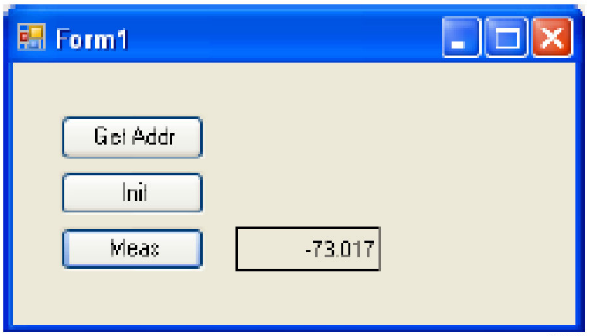
4. Disconnect and reconnect the instrument.
5. Wait for the operating system to find the new hardware.
6. Restart the Power Meter application.
7. Confirm that the instrument address shown at the top of the Power Meter window is the correct address.
8. Close the Power Meter application.

Make a CW Measurement

When creating an application for making a simple continuous wave (CW) measurement, a similar approach is taken whether you are using VB 6.0, VB.Net, or C#. The VB.NET and C# examples in this section were created using Microsoft Visual Studio 2005. Thfor taking a measurementis code assumes that a single instrument has been connected to the computer and has proven functional. If an earlier version of Visual Studio.NET is being used, the VB.NET and C# code may need minor modification, as a direct copy and paste may not work.

NOTE. *Before performing the following procedures, make sure you have installed the Sample Code application on your PC and that the instrument and PC are communicating properly. For installation and functional check procedures, see the Installation and Safety Instructions that shipped with your instrument. You can also download the manual from www.tektronix.com.manuals.*

1. Start the code by creating a default Windows application.
2. Place three buttons and one label on the window or form.
3. Name the buttons as follows:
   - **cmdGetAddress**: Clicking this button uses the "LB_GetAddress_Idx" command. The name of this command can be interpreted as "get the address using the index." In this case, the first instrument is used, or the instrument with an index of 1.
   - **cmdInitialize**: Clicking this button uses the "LB_InitializeSensor_Addr" command. The name of this command can be interpreted as "initialize the instrument using the address". Initialization causes the calibration constants and other information for the instrument to be transferred to the PC.
   - **cmdMeasure**: Clicking this button uses the "LB_CWMeasure" command. The name of this command can be interpreted as “make a measurement”. The result of the measurement is converted to text and placed in the label. This command requires the address acquired in the first button click. It also requires that the instrument be initialized, as done in the second button click. In this API, most commands are designed for use with the address.
4. Name the label **lblCW**. The result of the measurement taken when the cmdMeasure command is used is converted to text and placed in this label.
5. Copy the appropriate portions of code from the following pages.
6. Compile the application you have just coded. The window may look similar to the one shown here.



0651-006

7. Run the application as follows:

   a. Click the “Get Addr” or cmdGetAddress button.

   b. Click the “Init” or cmdInitialize button. Wait for the message indicating initialization is complete. This typically takes about 5 seconds.

   c. Click the “Meas” or cmdMeasure button and a measurement should appear in the label. Now that the instrument has been initialized, this button can be clicked repeatedly for as many measurements as you would like.

Tips

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#.
- The default ByRef/ByVal are switched when going from VB 6.0 to VB.NET and C#. 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#. VB.NET does have the following type of declaration, which can be used inside a structure. However, it seems able to be passed via a_stdcall for simple structures only.

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

NOTE. *If you are using an earlier version of Visual Studio.NET, the code may need some modification.*

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# 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###" );
            }
         }
    }
}
```

Command Groups

The following command group tables organize commands together by functionality and link each command to the individual command, located in the Commands Listed in Alphabetical Order section of this manual. (See page 22, *Commands Listed in Alphabetical Order*.)

If a command has related commands, the related commands are indented to the primary command in the table. Primary commands are listed in the Commands Listed in Alphabetical Order section, and in the table of contents.

NOTE. *Unless otherwise noted the following commands are valid for all instrument models.*

CW Measurement Command Group

Command	Description
LB_MeasureCW (See page 46, *LB_MeasureCW*.)	Makes continuous wave (CW) measurements. The value returned is in the units currently selected.
LB_MeasureCW_PF (See page 48, *LB_MeasureCW_PF* .)	Makes continuous wave (CW) measurements and evaluates the measurement relative to the current limit. The value returned is in the units currently selected.

Initialization and Identification Command Group

Command	Description
LB_AddressConflictExists (See page 23, *LB_AddressConflictExists*.)	Checks the address of all instruments that are connected to the system. If any of the addresses match, a conflict is deemed to exist. If all the addresses are unique to the system, a conflict is deemed not to exist.
LB_BlinkLED_Addr **LB_BlinkLED_Idx** **LB_BlinkLED_SN** (See page 24, *LB_BlinkLED_Addr (and related commands)*.)	Cause the instrument LED to blink four times.
LB_ChangeAddress (See page 26, *LB_ChangeAddress*.)	Changes the address of the device. The address is changed from “currentAddr” to “newAddr”.
LB_IsDeviceInUse_Addr **LB_IsDeviceInUse_Idx** **LB_IsDeviceInUse_SN** (See page 40, *LB_IsDeviceInUse_Addr (and related commands)*.)	Return a 1 if the device has been initialized and a 0 if the device has not been initialized by the calling program or any other program.
LB_DriverVersion (See page 27, *LB_DriverVersion*.)	Returns a 32 bit integer indicating the version of LB_API2.dll.
LB_GetFirmwareVersion (See page 28, *LB_GetFirmwareVersion*.)	Returns a null-terminated string of chars indicating the firmware version.

Command	Description
LB_GetIndex_Addr **LB_GetIndex_SN** (See page 29, *LB_GetIndex_Addr (and related commands)*.)	Return the index given to the serial number or address.
LB_GetModelNumber_Addr **LB_GetModelNumber_Idx** **LB_GetModelNumber_SN** (See page 31, *LB_GetModelNumber_Addr (and related commands)*.)	Return a value equating to a model number enumeration.
LB_GetSerNo_Addr **LB_GetSerNo_Idx** (See page 35, *LB_GetSerNo_Addr (and related commands)*.)	Return the serial number given the index or address.
LB_InitializeSensor_Addr **LB_InitializeSensor_Idx** **LB_InitializeSensor_SN** (See page 37, *LB_InitializeSensor_Addr (and related commands)* .)	Cause the instrument to be initialized.
LB_IsSensorConnected_Addr **LB_IsSensorConnected_SN** (See page 42, *LB_IsSensorConnected_Addr (and related commands)*.)	Determine if the specified instrument is connected. The query is based on the serial number or address.
LB_SensorCnt (See page 57, *LB_SensorCnt*.)	Returns the number of instruments currently connected to the computer.
LB_SetAddress_Idx **LB_GetAddress_Idx** (See page 61, *LB_SetAddress_Idx (and related commands)*.)	Return the address, given the index and vice versa. The index is assigned by the OS when the unit is plugged in.
LB_SetAddress_SN **LB_GetAddress_SN** (See page 63, *LB_SetAddress_SN (and related commands)*.)	Return the address, given the serial number and vice versa.
LB_WillAddressConflict (See page 120, *LB_WillAddressConflict*.)	Checks the address of all instruments connected to the system. If any of the addresses match, a conflict is deemed to exist.

Pulse Measurement Command Group

NOTE. *These commands are only valid for PSM4000 and PSM5000 Series instruments.*

Command	Description
LB_MeasureBurst_DBM (See page 44, *LB_MeasureBurst_DBM*.)	Measures the peak power, minimum power and average power over a specified measurement interval or burst. The measurement is made relative to a trigger.
LB_MeasurePulse (See page 51, *LB_MeasurePulse*.)	Makes pulse measurements. The measurement returns pulse power (average power in the pulse); peak power (highest sample measured); averge power; and duty cycle.
LB_MeasurePulse_PF (See page 53, *LB_MeasurePulse_PF*.)	Makes pulse measurements just as LBMeasurePulse does, except that the pulse power (instead of peak or average) is evaluated against the selected limit.

Pulse Setup Command Group

NOTE. *These commands are only valid for PSM4000 and PSM5000 Series instruments.*

Command	Description
LB_SetAutoPulseEnabled **LB_GetAutoPulseEnabled** (See page 67, *LB_SetAutoPulseEnabled (and related commands)* .)	Enable or disable the default or automatic pulse measurement criteria.
LB_SetPulseCriteria **LB_GetPulseCriteria** (See page 100, *LB_SetPulseCriteria (and related commands)* .)	Set or get the pulse measurement criteria.
LB_SetPulseReference **LB_GetPulseReference** (See page 103, *LB_SetPulseReference (and related commands)* .)	Configure the instrument for relative measurements during pulse measurements. (Other commands set a reference for CW measurements.)

Pulse Profiling Gate Command Group

NOTE. *These commands are valid only for the PSM5000 Series instrument.*

Command	Description
PP_GatePositionIsValid (See page 126, *PP_GatePositionIsValid* .)	Determines whether the specified gate is valid. The gate index may be 0..4.
PP_GetGateCrestFactor (See page 128, *PP_GetGateCrestFactor*.)	Returns the create factor (in dB) of the span in the analysis trace specified by the gate.
PP_GetGateDroop (See page 129, *PP_GetGateDroop*.)	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.
PP_GetGateDutyCycle (See page 130, *PP_GetGateDutyCycle* .)	Returns the duty cycle (as a decimal) of span in the analysis trace specified by the gate.
PP_GetGateEndPosition (See page 132, *PP_GetGateEndPosition* .)	Returns the location, as an index in the analysis trace, of the right side of the specified gate.
PP_GetGateFallTime (See page 133, *PP_GetGateFallTime* .)	Returns the fall time in microseconds of the pulse delineated by the selected gate.
PP_GetGateOverShoot (See page 135, *PP_GetGateOverShoot* .)	Returns the overshoot in dB.
PP_GetGatePeakPower (See page 136, *PP_GetGatePeakPower* .)	Returns the peak power measured of the analysis trace as defined by the gate edges.
PP_GetGatePRF (See page 137, *PP_GetGatePRF*.)	Returns the pulse repetition frequency (PRF) 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).
PP_GetGatePRT (See page 139, *PP_GetGatePRT*.)	Returns the pulse repetition time (PRT) in microseconds using the same algorithm defined for PRF. The sole difference is that time instead of frequency is returned.
PP_GetGatePulsePower (See page 141, *PP_GetGatePulsePower* .)	Returns average pulse power.
PP_GetGatePulseWidth (See page 143, *PP_GetGatePulseWidth* .)	Measures the pulse width in microseconds.
PP_GetGateRiseTime (See page 145, *PP_GetGateRiseTime* .)	Returns rise time in microseconds.
PP_SetGateStartEndPosition **PP_SetGateStartEndTime** **PP_GetGateStartEndTime** **PP_SetGateStartPosition** **PP_GetGateStartPosition** **PP_SetGateEndPosition** **PP_GetGateEndPosition** **PP_SetGateStartTime** **PP_GetGateStartTime** **PP_SetGateEndTime** **PP_GetGateEndTime** (See page 179, *PP_SetGateStartEndPosition (and related commands)*.)	Sets or gets the gate start (left side) and/or end (right side) in terms of trace index or time.

Pulse Profiling Marker Command Group

NOTE. *These commands are valid only for the PSM5000 Series instrument.*

Command	Description
PP_GetMarkerAmp (See page 146, *PP_GetMarkerAmp* .)	Returns the amplitude of the trace at the point indicated by the marker.
PP_GetMarkerDeltaAmp (See page 147, *PP_GetMarkerDeltaAmp* .)	Returns the difference in amplitude between the normal marker and the delta marker in dBm.
PP_MarkerPosIsValid (See page 162, *PP_MarkerPosIsValid*.)	Returns the state of the selected marker.
PP_MarkerToPk **PP_MarkerToLowestPk** **PP_MarkerToFirstPk** **PP_MarkerToLastPk** **PP_MarkerPrevPk** **PP_MarkerNextPk** **PP_MarkerPkHigher** **PP_MarkerPkLower** (See page 163, *PP_MarkerToPk (and related commands)* .)	Set one of five markers (0<=mrkIdx<=4) to the position specified in the command.
PP_SetMarkerDeltaTime **PP_GetMarkerDeltaTime** (See page 184, *PP_SetMarkerDeltaTime (and related commands)* .)	Sets or gets the positions the selected marker in microseconds.
PP_SetMarkerMode **PP_GetMarkerMode** (See page 186, *PP_SetMarkerMode (and related commands)* .)	Sets or gets the marker mode to on, normal or delta marker.
PP_SetMarkerPosition **PP_GetMarkerPosition** **PP_SetMarkerPositionTime** **PP_GetMarkerPositionTime** (See page 188, *PP_SetMarkerPosition (and related commands)*.)	Sets or gets the position of the normal or delta marker depending on the marker mode.

Pulse Profiling Setup Command Group

NOTE. These commands are valid only for the PSM5000 Series instrument.

Command	Description
PP_GetPulseEdgesTime **PP_GetPulseEdgesPosition** (See page 152, *PP_GetPulseEdgesTime (and related commands)*.)	Return the index of the leading and trailing edges of the pulse containing the peak defined by pkTime or pkIdx.
PP_SetAvgMode **PP_GetAvgMode** **PP_GetTraceAvgs** **PP_ResetTraceAveraging** (See page 169, *PP_SetAvgMode (and related commands)*.)	Set, auto-set or manual reset the averaging mode.
PP_SetAvgResetSens **PP_GetAvgResetSens** (See page 172, *PP_SetAvgResetSens (and related commands)*.)	Set or get the criteria used to reset the averaging when the averaging mode is AVG_AUTO_RESET (see PP_SetAvgMode and PP_GetAvgMode).
PP_SetClosestSweepTimeUSEC (See page 173, *PP_SetClosestSweepTimeUSEC*.)	Sets the sweep time to the fixed sweep time closest to the sweep time sent (in microseconds) to the command.
PP_SetFilter **PP_GetFilter** (See page 174, *PP_SetFilter (and related commands)* .)	Sets or gets the enumeration associated with the current filter settings.
PP_SetGateMode **PP_GetGateMode** (See page 177, *PP_SetGateMode (and related commands)* .)	Sets or gets the gate mode.
PP_SetMeasurementThreshold **PP_GetMeasurementThreshold** (See page 190, *PP_SetMeasurementThreshold (and related commands)*.)	Sets or gets the measurement threshold. The measurement threshold, along with the peak criteria, affects a number of measurement commands, especially the peak commands.
PP_SetPoles **PP_GetPoles** (See page 192, *PP_SetPoles (and related commands)* .)	Sets or gets the number of poles in the current filter.
PP_SetSweepDelay **PP_GetSweepDelay** (See page 194, *PP_SetSweepDelay (and related commands)*.)	Sets or gets the sweep delay in microseconds. Sweep delay is the time between the trigger and the start of data acquisition.

Command	Description
PP_SetSweepDelayMode **PP_GetSweetDelayMode** (See page 196, *PP_SetSweepDelayMode (and related commands)*.)	Turns the sweep delay on or off. The sweep delay parameter remains unchanged.
PP_SetSweepHoldOff **PP_GetSweepHoldOff** (See page 197, *PP_SetSweepHoldOff (and related commands)*.)	Specifies the length of time (in microseconds) to wait after a sweep or trace is taken.
PP_SetSweepTime **PP_GetSweepTime** (See page 198, *PP_SetSweepTime (and related commands)*.)	Sets or gets the sweep time (in microseconds) for the next sweep taken. Sweep time is a 1, 2, 5 sequence starting with 10 μsec and ending with 1 second.
PP_SetTimeOut **PP_GetTimeOut** (See page 200, *PP_SetTimeOut (and related commands)*.)	Sets or gets the timeout used while taking a trace.

Pulse Profiling Status Command Group

NOTE. *These commands are valid only for the PSM5000 Series instrument.*

Command	Description
PP_AnalysisTraceIsValid (See page 121, *PP_AnalysisTraceIsValid* .)	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.
PP_CheckTrigger (See page 122, *PP_CheckTrigger* .)	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.

Pulse Profiling Trace Command Group

NOTE. *These commands are valid only for the PSM5000 Series instrument.*

Command	Description
PP_CnvtTrace (See page 123, *PP_CnvtTrace*.)	Converts a trace (trIn) from one unit to another, and stores the converted values in a new trace (trOut).
PP_CurrTrace2AnalysisTrace (See page 125, *PP_CurrTrace2AnalysisTrace* .)	Copies the current trace to the analysis trace and returns a copy of that trace.

Command	Description
PP_GetAnalysisTraceLength (See page 127, *PP_GetAnalysisTraceLength* .)	Returns a 32 bit integer indicating the length of the analysis trace.
PP_GetPeaks_Val **PP_GetPeaks_Idx** **PP_GetPeaksFromTr_Val** **PP_GetPeaksFromTr_Idx** **PP_GetPeaks_VEE_Idx** **PP_GetPeaks_VEE_Val** (See page 148, *PP_GetPeaks_Val (and related commands)* .)	Return a set of peaks from either the analysis trace (PP_GetPeaks_Val and PP_GetPeaks_Idx) or from a trace passed to the command.
PP_GetPulseEdgesTime **PP_GetPulseEdgesPosition** (See page 152, *PP_GetPulseEdgesTime (and related commands)*.)	Returns the index of the leading and trailing edges of the pulse containing the peak defined by pkTime or pkIdx.
PP_GetTrace (See page 155, *PP_GetTrace*.)	Causes the instrument to take a trace and return the resultant data. The trace is an array of equally spaced (in time) samples, in dBm.
PP_GetTraceAvgPower **PP_GetTraceCrestFactor** **PP_GetTraceDC** **PP_GetTracePkPwr** **PP_GetTracePulsePower** (See page 158, *PP_GetTraceAvgPower (and related commands)* .)	Make a number of measurements similar to the power meter measurement commands, but operate on a single trace instead of a set of random samples.
PP_GetTraceLength (See page 160, *PP_GetTraceLength*.)	Returns the number of trace points associated with the current sweep time.
PP_SetAnalysisTrace **PP_GetAnalysisTrace** (See page 167, *PP_SetAnalysisTrace (and related commands)*.)	Enable the user to get and set the analysis trace directly. These are companions to PP_CurrTrace2AnalysisTrace.

Pulse Profiling Trigger Command Group

NOTE. *These commands are valid only for the PSM5000 Series instrument.*

Command	Description
PP_SetTriggerEdge **PP_GetTriggerEdge** (See page 201, *PP_SetTriggerEdge (and related commands)*.)	Sets or gets the trigger signal edge on which the beginning of the trace will occur. The values are positive edge or negative edge.
PP_SetTriggerLevel **PP_GetTriggerLevel** (See page 203, *PP_SetTriggerLevel (and related commands)* .)	Sets or gets the trigger level for internal triggering (manual or automatic), in dBm.
PP_SetTriggerOut **PP_GetTriggerOut** (See page 205, *PP_SetTriggerOut (and related commands)*.)	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).
PP_SetTriggerSource **PP_GetTriggerSource** (See page 207, *PP_SetTriggerSource (and related commands)*.)	Sets or gets the trigger source. Trigger source can be internal or external.

Save/Recall Command Group

Command	Description
LB_ReadStateFromINI **LB_WriteStateToINI** (See page 56, *LB_ReadStateFromINI (and related commands)*.)	Cause the current state, including all numbered registers to be written to an INI file.
LB_ResetRegStates **LB_ResetCurrentState** (See page 55, *LB_ResetRegStates (and related commands)*.)	Enable the user to cause either the current state or the state information held in the save/recall registers to be reset.
LB_StoreReg **LB_RecallReg** (See page 118, *LB_StoreReg (and related commands)*.)	Store/recall register commands. There are 20 registers and each register holds an entire state.

Service Command Group

Command	Description
LB_SetCalDueDate (See page 71, *LB_SetCalDueDate (and related commands)* .)	Set or get the calibration due date, which is specified by serial number, not address.

Setup Command Group

Command	Description
LB_Set75OhmsEnabled **LB_Get75OhmsEnabled** (See page 60, *LB_Set75OhmsEnabled (and related commands)*.)	Used to correct measurements made using a minimum loss pad.
LB_SetAntiAliasingEnabled **LB_GetAntiAliasingEnabled** (See page 65, *LB_SetAntiAliasingEnabled (and related commands)*.)	Enable or disable the anti-aliasing feature or allow its state to be checked.
LB_SetAverages **LB_GetAverages** (See page 69, *LB_SetAverages (and related commands)*.)	Set or get the number of data buffers that are averaged. The default is set to 75.
LB_SetCWReference **LB_GetCWReference** (See page 74, *LB_SetCWReference (and related commands)*.)	Set up the instrument for relative measurements during CW measurements.
LB_SetDutyCycleEnabled **LB_GetDutyCycleEnabled** **LB_SetDutyCyclePerCent** **LB_GetDutyCyclePerCent** (See page 76, *LB_SetDutyCycleEnabled (and related commands)*.)	Set up the instrument for average pulse power measurements based on an assumed duty cycle value.
LB_SetExtendedAveraging **LB_GetExtendedAveraging** **LB_SetExtendedAveragingEnabled** **LB_GetExtendedAveragingEnabled** **LB_GetExtendedAveragingEnabled** **LB_ResetExtendedAveraging** (See page 79, *LB_SetExtendedAveraging (and related commands)*.)	Manage the extended average settings of the instrument. Extended averaging uses a form of exponential averaging.

Command	Description
LB_SetFrequency **LB_GetFrequency** (See page 82, *LB_SetFrequency (and related commands)*.)	Set or get the frequency of the addressed device, in Hz.
LB_SetLimitEnabled **LB_GetLimitEnabled** **LB_SetSingleSidedLimit** **LB_GetSingleSidedLimit** **LB_SetDoubleSidedLimit** **LB_GetDoubleSidedLimit** (See page 84, *LB_SetLimitEnabled (and related commands)*.)	Set and get limits, and specify single-sided limits, double- sided limits, or neither limit.
LB_SetMaxHoldEnabled **LB_GetMaxHoldEnabled** **LB_ResetMaxHold** (See page 93, *LB_SetMaxHoldEnabled (and related commands)* .)	Cause CW or pulse measurements to retain the greater of the most recent measurement or previous measurements.
LB_SetMeasurementPowerUnits **LB_GetMeasurementPowerUnits** (See page 95, *LB_SetMeasurement-PowerUnits (and related commands)* .)	Set or get the measurement power units.
LB_SetOffset **LB_GetOffset** **LB_SetEnabled** **LB_GetEnabled** (See page 97, *LB_SetOffset (and related commands)*.)	Cause a fixed offset to be added to the reading, or enable/disable the feature.
LB_SetResponseEnabled **LB_GetResponseEnabled** **LB_SetResponse** **LB_GetResponse** (See page 106, *LB_SetResponseEnabled (and related commands)*.)	Set the response, which is a frequency sensitive offset, and enable/disable the feature.

Trigger Command Group

Command	Description
LB_SetTTLTriggerInEnabled **LB_GetTTLTriggerInEnabled** **LB_SetTTLTriggerInInverted** **LB_GetTTLTriggerInInverted** **LB_SetTTLTriggerInTimeOut** **LB_GetTTLTriggerInTimeOut** (See page 110, *LB_SetTTLTriggerInEnabled (and related commands)*.)	Control or read back the state of the external trigger input. The trigger-in can be enabled, disabled or inverted; or the timeout value can be set or read.
LB_SetTTLTriggerOutEnabled **LB_GetTTLTriggerOutEnabled** **LB_SetTTLTriggerOutInverted** **LB_GetTTLTriggerOutInverted** (See page 114, *LB_SetTTLTriggerOutEnabled (and related commands)*.)	Control the trigger output of the device. It can be enabled, disabled, inverted or normal.

Commands Listed in Alphabetical Order

The following commands are exported from a Visual C++ 2005 project. The calling convention used is _stdcall. The declarations are shown in various examples for C++, C#, VB 6.0, and/or VB.NET.

The driver is installed using a .inf file. The supplied apausb.inf file is in the Tektronix\Tektronix Power Sensor Applications directory of the install directory.

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

C#	LB_API2_Declarations.cs
VB.NET	LB2_Declarations.vb

NOTE. *Pre-allocated buffers are often required when strings (such as serial number) or arrays are being passed back from the driver by reference (or pointer).*

LB_AddressConflictExists

This command checks the address of all instruments that are connected to the system. If any of the addresses match, a conflict is deemed to exist. If all the addresses are unique to the system, a conflict is deemed not to exist.

Pass Parameters:

None

Returned Values:

Conflict Exists = 1

Conflict does not exit = 0

Error < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++	LB_API2 long _stdcall LB_AddressConflictExists();
VB 6.0	Public Declare Function LB_AddressConflictExists _ Lib "LB_API2.dll" () _ As Long
VB.NET	Public Declare Function LB_AddressConflictExists _ Lib "LB_API2.dll" () _ As Integer
C#	[System.Runtime.InteropServices.DllImport("LB_API2.dll")] public static extern int LB_AddressConflictExists();

LB_BlinkLED_Addr (and related commands)

Related Commands:

LB_BlinkLED_Idx

LB_BlinkLED_SN

These commands cause the instrument LED to blink four times. This is intended to allow the user to ID the instrument physically.

Pass Parameters:

Index, serial number or address

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_BlinkLED_SN(char* SN);

LB_API2 long _stdcall LB_BlinkLED_Idx(long idx);

LB_API2 long _stdcall LB_BlinkLED_Addr(long addr);
```

VB 6.0

```
Public Declare Function LB_BlinkLED_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Long

Public Declare Function LB_BlinkLED_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Long) _
    As Long

Public Declare Function LB_BlinkLED_Addr _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long) _
    As Long
```

VB.NET

```
Public Declare Function LB_BlinkLED_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Integer

Public Declare Function LB_BlinkLED_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Integer) _
    As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_BlinkLED_SN(
    string sn );

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

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

LB_ChangeAddress

This command changes the address of the device. The address is changed from “currentAddr” to “newAddr”. The address is retained in non-volatile memory.

Pass Parameters:

currentAddr = 1 to 255

newAddr = 1 to 255

Returned Values:

Success: > 0

Failure: < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group.*)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_ChangeAddress(long currentAddr, long newAddr);
```

VB 6.0

```
Public Declare Function LB_ChangeAddress _
    Lib "LB_API2.dll" ( _
    ByVal currentAddr As Long, _
    ByVal newAddr As Long) _
    As Long
```

VB.NET

```
Public Declare Function LB_ChangeAddress _
    Lib "LB_API2.dll" ( _
    ByVal currentAddr As Integer, _
    ByVal newAddr As Integer) _
    As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_ChangeAddress(
    int currentAddr,
    int newAddr );
```

LB_DriverVersion

This command returns a 32 bit integer indicating the version of LB_API2.dll.

Pass Parameters:

None

Returned Values:

The least two significant digits indicate the revision number. The next two least significant digits represent the minor version. The most significant digit(s) represent the major version. As of this writing, the value returned is 45030, so that:

- 30 is the revision,
- 50 is the minor version,
- 4 is the major version.

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++	Long LB_DriverVersion(void)
VB.NET	Public Declare Function LB_DriverVersion Lib “LB_API2.dll” () as Integer
C#	[System.Runtime.InteropServices.DllImport("LB_API2.dll")] public static extern int LB_DriverVersion();

LB_GetFirmwareVersion

This command returns a null-terminated string of chars indicating the firmware version.

Pass Parameters:

- addr is a 32 bit integer containing the address of the device.
- buff is a pointer to an array or buffer of chars. The characters must be allocated by the calling command. The buffer must be at least 14 chars long and should contain the value zero.
- buffLen indicates the length of buff. The length of buff must be at least 14 chars.

Returned Values:

A return value of greater than zero indicates success. A return value of less than zero indicates failure. The version information is returned in buff. The current value of buff is “1.28 07/08/08”. The first part (1.28) indicates the major and minor version. The second part indicates the date.

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++	Long LB_GetFirmwareVersion(long addr, char* buff, long buffLen);
VB.NET	Public Declare Function LB_GetFirmwareVersion Lib "LB_API2.dll" (ByVal addr As Integer, ByRef buff As Byte, ByVal buffLen As Integer) As Integer
C#	[System.Runtime.InteropServices.DllImport("LB_API2.dll")] public static extern int LB_GetFirmwareVersion(int addr, ref byte buff, int buffLen);

LB_GetIndex_Addr (and related commands)

Related Commands:

LB_GetIndex_SN

These commands return the index given to the serial number or address.

Pass Parameters:

Serial number or address

Returned Values:

Success: > Index greater than 0

Error: <= 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_GetIndex_SN(char* SN);

LB_API2 long _stdcall LB_GetIndex_Addr(long addr);
```

VB 6.0

```
Public Declare Function LB_GetIndex_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Long

Public Declare Function LB_GetIndex_Addr _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long) _
    As Long
```

VB.NET

```
Public Declare Function LB_GetIndex_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetIndex_SN(
    string sn );

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

LB_GetModelNumber_Addr (and related commands)

Related Commands:

LB_GetModelNumber_Idx

LB_GetModelNumber_SN

These commands return a value equating to a model number enumeration. In each case, the serial number, index or address must be passed. The suffix (_SN, _Idx or Addr) denotes the type of pass parameter.

Pass Parameters:

Serial number, address or index.

Returned Values:

Returned values are -1 to 113 as denoted by the enumerations below.

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
enum MODEL_NUMBER {
    Unknown = -1,
    PSM4110 = 101,
    PSM4120 = 102,
    PSM5110 = 103,
    PSM5120 = 104,
    PSM3110 = 105,
    PSM3120 = 106,
    PSM3310 = 107,
    PSM3320 = 108,
    PSM3510 = 109,
    PSM4410 = 110,
    PSM4320 = 111,
    PSM5410 = 112,
    PSM5320 = 113
};

LB_API2 long _stdcall LB_GetModelNumber_SN(char* SN,
                MODEL_NUMBER* modelNumber);

LB_API2 long _stdcall LB_GetModelNumber_Idx(long idx,
                MODEL_NUMBER* modelNumber);

LB_API2 long _stdcall LB_GetModelNumber_Addr(long addr,
                MODEL_NUMBER* modelNumber);
```

VB 6.0

```
Public Enum MODEL_NUMBER ' Enumeration of model numbers
    Unknown = -1
    PSM4110 = 101
    PSM4120 = 102
    PSM5110 = 103
    PSM5120 = 104
    PSM3110 = 105
    PSM3120 = 106
    PSM3310 = 107
    PSM3320 = 108
    PSM3510 = 109
    PSM4410 = 110
    PSM4320 = 111
    PSM5410 = 112
    PSM5320 = 113
End Enum

Public Declare Function LB_GetModelNumber_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String, _
    ByRef modelNumber As MODEL_NUMBER) _
    As Long

Public Declare Function LB_GetModelNumber_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Long, _
    ByRef modelNumber As MODEL_NUMBER) _
    As Long

Public Declare Function LB_GetModelNumber_Addr _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef modelNumber As MODEL_NUMBER) _
    As Long
```

VB.NET

```
Public Enum MODEL_NUMBER
    Unknown = -1
    PSM4110 = 101
    PSM4120 = 102
    PSM5110 = 103
    PSM5120 = 104
    PSM3110 = 105
    PSM3120 = 106
    PSM3310 = 107
    PSM3320 = 108
    PSM3510 = 109
    PSM4410 = 110
    PSM4320 = 111
    PSM5410 = 112
    PSM5320 = 113
End Enum
```

```
Public Declare Function LB_GetModelNumber_SN _
       Lib "LB_API2.dll" ( _
       ByVal sn As String, _
       ByRef modelNumber As MODEL_NUMBER) _
       As Integer

Public Declare Function LB_GetModelNumber_Idx _
       Lib "LB_API2.dll" ( _
       ByVal idx As Integer, _
       ByRef modelNumber As MODEL_NUMBER) _
       As Integer

Public Declare Function LB_GetModelNumber_Addr _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef modelNumber As MODEL_NUMBER) _
       As Integer
```

C#

```
public enum MODEL_NUMBER
{   // Enumeration of model numbers
       Unknown = -1,
       PSM4110 = 101,
       PSM4120 = 102,
       PSM5110 = 103,
       PSM5120 = 104,
       PSM3110 = 105,
       PSM3120 = 106,
       PSM3310 = 107,
       PSM3320 = 108,
       PSM3510 = 109,
       PSM4410 = 110,
       PSM4320 = 111,
       PSM5410 = 112,
       PSM5320 = 113
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetModelNumber_SN(
       string sn,
       ref MODEL_NUMBER modelNumber);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetModelNumber_Idx(
       int idx,
       ref MODEL_NUMBER modelNumber);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetModelNumber_Addr(
       int addr,
       ref MODEL_NUMBER modelNumber);
```

LB_GetSerNo_Addr (and related commands)

Related Commands:

LB_GetSerNo_Idx

These commands return the serial number given the index or address. These and other similar commands require a pre-allocated buffer.

Pass Parameters:

Index or address

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_GetSerNo_Idx(long idx, char* SN);
LB_API2 long _stdcall LB_GetSerNo_Addr(long addr, char* SN);
```

VB 6.0

```
Public Declare Function LB_GetSerNo_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Long, _
    ByVal sn As String) _
    As Long

Public Declare Function LB_GetSerNo_Addr _
    Lib "LB_API2.dll" ( _
    ByVal address As Long, _
    ByVal sn As String) _
    As Long
```

VB.NET

```
Public Declare Function LB_GetSerNo_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Integer, _
    ByVal sn As String) _
    As Integer

Public Declare Function LB_GetSerNo_Addr _
    Lib "LB_API2.dll" ( _
    ByVal address As Integer, _
    ByVal sn As String) _
    As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetSerNo_Idx( int idx, string sn );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetSerNo_Addr( int address, string sn );
```

LB_InitializeSensor_Addr (and related commands)

Related Commands:

LB_InitializeSensor_Idx

LB_InitializeSensor_SN

These commands cause the instrument to be initialized. This includes downloading the calibration factors and other data required to operate the instrument. Initialization normally takes about five seconds.

Pass Parameters:

Index, serial number or address

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_InitializeSensor_SN(char* SN);

LB_API2 long _stdcall LB_InitializeSensor_Idx(long idx);

LB_API2 long _stdcall LB_InitializeSensor_Addr(long addr);
```

VB 6.0

```
Public Declare Function LB_InitializeSensor_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Long

Public Declare Function LB_InitializeSensor_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Long) _
    As Long

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

VB.NET

```
Public Declare Function LB_InitializeSensor_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Integer

Public Declare Function LB_InitializeSensor_Idx _
    Lib "LB_API2.dll" ( _
    ByVal idx As Integer) _
    As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_InitializeSensor_SN(
    string sn );

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

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

LB_IsDeviceInUse_Addr (and related commands)

Related Commands:

LB_IsDeviceInUse_Idx

LB_IsDeviceInUse_SN

These commands return a 1 if the device has been initialized and a 0 if the device has not been initialized by the calling program or any other program. These functions are intended to be used in multi-threaded applications. Therefore, if an instrument has already been initialized, it can be an indication that the instrument is in use by another application.

Pass Parameters:

addr – address of the device

idx – the index of the device (generally not known)

SN – the serial number of the device

Returned Values:

1 if the device has been initialized, and a 0 if the device has not been initialized

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
long LB_IsDeviceInUse_Idx(long idx);
long LB_IsDeviceInUse_Addr(long addr);
long LB_IsDeviceInUse_SN(char* SN);
```

VB.NET

```
Public Declare Function LB_IsDeviceInUse_Idx Lib "LB_API2.dll"
       (ByVal idx As Integer)
      As Integer

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

Public Declare Function LB_IsDeviceInUse_SN Lib "LB_API2.dll"
       (ByVal sn As String)
       As Integer
```

C#

```
int LB_IsDeviceInUse_Idx(int idx);
int LB_IsDeviceInUse_Addr(int addr);
int LB_IsDeviceInUse_SN(ref byte SN);
```

LB_IsSensorConnected_Addr (and related commands)

Related Commands:

LB_IsSensorConnected_SN

These commands determine if the specified instrument is connected. The query is based on the serial number or address. The omission of “LB_IsSensorConnected_Idx” is not an error. The LB_SensorCnt() does the job more simply and directly.

Pass Parameters:

Serial number or address.

Returned Values:

Serial Number is connected: 1

Serial Number is NOT connected: 0

Error < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_IsSensorConnected_SN(char* SN);
LB_API2 long _stdcall LB_IsSensorConnected_Addr(long addr);
```

VB 6.0

```
Public Declare Function LB_IsSensorConnected_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Long

Public Declare Function LB_IsSensorConnected_Addr _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long) _
    As Long
```

VB.NET

```
Public Declare Function LB_IsSensorConnected_SN _
    Lib "LB_API2.dll" ( _
    ByVal sn As String) _
    As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_IsSensorConnected_SN(
    string sn );

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

LB_MeasureBurst_DBM

This command measures the peak power, minimum power and average power over a specified measurement interval or burst. The measurement is made relative to a trigger. The trigger is either an internal automatic trigger or and external TTL trigger. The internal trigger triggers on the rising edge of the first pulse. The measurement starts after a specified delay and continues for the specified measurement time. Both are specified in microseconds with a resolution (or increments) or 2 microseconds.

Pass Parameters:

addr – address of the device

delayUSEC – the delay of the measurement relative to the trigger (0, 2, 4… µs)

measTimeUSEC – the measurement time (2, 4, 6 … µs)

trgInt – determines if the trigger is internal-automatic or external TTL

pk – returns the maximum or peak value encountered during the measurement time.

avg – returns the average power during the measurement time

min – returns the minimum power level encountered during the measurement time

NOTE. *delayUSEC + measTimeUSEC must be less than 1,000,000 µs or 1 second.*

Returned Values:

Success: > 1

Failure: <= 0

Command Group:

Pulse Measurement

(See page 12, *Pulse Measurement Command Group*.)

Sample Code Declarations:

C++

```
long LB_MeasureBurst_DBM(long addr, long delayUSEC, long measTimeUSEC, long trgInt, double* pk,
double *avg, double *min);
```

VB.NET

```
Public Declare Function LB_MeasureBurst_DBM Lib "LB_API2.dll" _
    (ByVal addr As Integer, _
    ByVal delayUSEC As Integer, _
    ByVal measTimeUSEC As Integer, _
    ByVal trgInt As Integer, _
    ByRef pk As Double, _
    ByRef avg As Double, _
    ByRef min As Double) As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_MeasureBurst_DBM(
    int addr,
    int delayUSEC,
    int measTimeUSEC,
    int trgInt,
    ref double pk,
    ref double avg,
    ref double min);
```

LB_MeasureCW

This command makes CW measurements. The value returned is in the units currently selected. The time to make this measurement can vary widely. Measurement time in particular depends on the setting of averaging. Typical measurement times are about 0.3 to 1.0 ms per buffer. Each buffer contains about 120 averages so that a measurement for 100 buffers (averaging set to 100) would take 30 to 100 ms. Another setting that affects the measurement time is anti-aliasing. The measurement time is about 40% greater with anti-aliasing on than with anti-aliasing off. Anti-aliasing is generally required if the baseband content (or demodulated signal) has a frequency above 200 kHz. Finally, getting an accurate measurement requires that the frequency be set.

Other commands that may be of interest are:

- LB_SetFrequency
- LB_GetFrequency
- LB_SetMeasurementPowerUnits
- LB_GetMeasurementPowerUnits
- LB_SetAntiAliasingEnabled
- LB_GetAntiAliasingEnabled
- LB_SetAverages
- LB_GetAverages

Pass Parameters:

Address and CW

Returned Values:

Success: > 0

Error: <= 0

Command Group:

CW Measurement

(See page 10, *CW Measurement Command Group*.)

Sample Code Declarations:

C++	`LB_API2 long _stdcall LB_MeasureCW(long addr, double* CW);`

VB 6.0	`Public Declare Function LB_MeasureCW _`
	`Lib "LB_API2.dll" ( _`
	`ByVal addr As Long, _`
	`ByRef CW As Double) As Long`

VB.NET

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

C#

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

LB_MeasureCW_PF

This command makes CW measurements, and evaluates the measurement relative to the current limit. The value returned is in the units currently selected. The measurement time affects for LB_MeasureCW_PF are the same as LB_MeasureCW.

Additionally, the limits should be set up. There are two types of limits: single-sided limits and double-sided limits. If the limits are set in one unit and the measurement is taken in another unit, the units are converted to a common base unit and then a comparison is made.

Other commands of interest in addition to the commands listed in LB_MeasureCW are:

- LB_SetLimitEnabled
- LB_SetSingleSidedLimit
- LB_SetDoubleSidedLimit
- LB_GetSingleSidedLimit
- LB_GetDoubleSidedLimit

Pass Parameters:

Address, CW and a PASS_FAIL_RESULT (long)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

CW Measurement

(See page 10, *CW Measurement Command Group*.)

Sample Code Declarations:

C++

```
enum PASS_FAIL_RESULT
{
    PASS= 0,                              // pass, measured value within limits
    FAIL_LOW= 1,                          // failed, measured value too low
    FAIL_HIGH= 2,                         // failed, measured value too high
    FAIL_BETWEEN_LIMIT_EXC= 3,            // failed between limits
    FAIL_BETWEEN_LIMIT_INC= 4,            // failed between limits
    NO_DETERMINATION = 5                  // no determination made,
};                                        // possible reasons include but are
                                          // not limited to the following reasons:
                                          //    - limits are not enabled
                                          //    - limits unspecified at freq
                                          //    - measurement not made
LB_API2 long _stdcall LB_MeasureCW_PF(
    long addr,
    double* CW,
    PASS_FAIL_RESULT* PF);
```

VB 6.0

```
Public Enum PASS_FAIL_RESULT
PASS = 0
    FAIL_LOW = 1                          ' pass measured value within limits
    FAIL_HIGH = 2                         ' failed measured value too low
    FAIL_BETWEEN_LIMIT_EXC = 3            ' failed measured value too high
    FAIL_BETWEEN_LIMIT_INC = 4            ' failed between limits
    NO_DETERMINATION = 5                  ' failed between limits
                                          ' no determination made; possible reasons include
                                           ' but are not limited to the following reasons:
                                           '     - limits are not enabled
                                           '     - limits are not specified
End Enum                                   '     - valid measurement not made (timeout?)

Declare Function LB_MeasureCW_PF _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef CW As Double, _
    ByRef pf As PASS_FAIL_RESULT) _
    As Long
```

VB.NET

```
Public Enum PASS_FAIL_RESULT
    PASS = 0                             ' pass measured value within limits
    FAIL_LOW = 1                         ' failed measured value too low
    FAIL_HIGH = 2                        ' failed measured value too high
    FAIL_BETWEEN_LIMIT_EXC = 3           ' failed between limits
    FAIL_BETWEEN_LIMIT_INC = 4           ' failed between limits
    NO_DETERMINATION = 5                 ' no determination made possible reasons include
                                          ' but are not limited to the following reasons:
                                          '     - limits are not enabled
                                          '     - limits are not specified
                                          '     - valid measurement not made (timeout?)
End Enum

Public Declare Function LB_MeasureCW_PF _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef CW As Double, _
    ByRef pf As PASS_FAIL_RESULT) _
    As Integer
```

C#

```
public enum PASS_FAIL_RESULT
{
    PASS = 0,
    FAIL_LOW = 1,
    FAIL_HIGH = 2,
    FAIL_BETWEEN_LIMIT_EXC = 3,
    FAIL_BETWEEN_LIMIT_INC = 4,
    NO_DETERMINATION = 5,
    // not limited to the following reasons:
    // - limits are not enabled
    // - limits are not specified
    // - valid measurement not made (timeout?)
}

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

LB_MeasurePulse

This command makes pulse measurements. The measurement returns pulse power (average power in the pulse); peak power (highest sample measured); averge power; and duty cycle. These are direct measurements. They are made using the number of buffers (averages) and the units specified using the LB_SetMeasurementPowerUnits command (See page 95, *LB_SetMeasurementPowerUnits (and related commands)* .). Much of the test that applies to CW measurement time also applies to pulse measurements.

***NOTE.** When using this command, the duty cycle is measured, it is not calculated. This is in contrast to the LB_SetDutyCycle commands that rely on an assumed duty cycle for making pulse power measurements.*

Since this command produces direct pulse measurements, if the stimulus parameters change (duty cycle, peak power or pulse power) the reading returned by this measurement will also change.

There are a number of items that can affect these measurements. One is the pulse peak criteria. Pulse peak criteria is relative to measured peak value. The changes will affect the duty cycle and pulse power. The affects will be most pronounced for pulses that have sloped rising and falling edges.

While peak measurement results can be obtained as low as -60dBm (or less), and at rates as fast as 3MHz with pulse widths less than 250 μs, the best measurements require some care. For best results, make pulse measurements when the pulse power is about 6dB above the peak noise, with averages set from about 50 to 100.

The best way to determine peak noise is to make a peak measurement with the signal off, and then examine the peak power readings. It is optimal if the pulse measurement is 6dB higher than the peak power with the power turned off - and other limits are not breached.

Finally, as the duty cycle decreases, averaging will need to be increased; and as PRF increases, the number of averages can be decreased. A good starting point is about 100 buffers or averages for a PRF of 10 kHz and a duty cycle of 10%. Adjust the averages inversely proportional to PRF and duty cycle - so if PRF doubles, you might be able to cut the averages by half. However, as a rule of thumb, it is a good idea to keep the number of averages above 50.

Other commands of interest in addition to the commands listed in LB_MeasurePulse are:

- LB_SetAutoPulseEnabled
- LB_GetAutoPulseEnabled
- LB_SetPulseCriteria
- LB_GetPulseCriteria

Pass Parameters:

Address, pulse power, peak power, average power and duty cycle. The last four elements should be provided by reference.

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Pulse Measurement

(See page 12, *Pulse Measurement Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_MeasurePulse(long addr,
        double* pulse,
        double* peak,
        double* average,
        double* dutyCycle);
```

VB 6.0

```
Public Declare Function LB_MeasurePulse _
        Lib "LB_API2.dll" (
        ByVal addr As Long, ByRef pulse As Double,
        ByRef peak as Double,
        ByRef average as Double,
        ByRef dutyCycle as Double)
        As Long
```

VB.NET

```
Public Declare Function LB_MeasurePulse _
        Lib "LB_API2.dll" ( _
        ByVal addr As Integer, _
        ByRef pulse As Double, _
        ByRef peak As Double, _
        ByRef average As Double, _
        ByRef dutyCycle As Double) As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_MeasurePulse(
        int addr,
        ref double pulse,
        ref double peak,
        ref double average,
        ref double dutyCycle );
```

LB_MeasurePulse_PF

This command makes pulse measurements just as LBMeasurePulse does. This is coupled with a pass/fail judgement like the LB_MeasureCW_PF function. The only difference is that the pulse power (instead of peak or average) is evaluated against the selected limit. Refer to the CW and Pulse measurement descriptions for more information.

Other commands of interest:

- LB_MeasureCW
- LB_MeasureCW_PF
- LB_MeasurePulse

Pass Parameters:

Address, pulse power, peak power, average power, duty cycle and pass/fail. The last five elements should be provided by reference or pointer.

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Pulse Measurement

(See page 12, *Pulse Measurement Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_MeasurePulse_PF(long addr,
    double* pulse,
    double* peak,
    double* average,
    double* dutyCycle,
    PASS_FAIL_RESULT* PF);
```

VB 6.0

```
Public Declare Function LB_MeasurePulse_PF _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef pulse As Double, _
    ByRef peak As Double, _
    ByRef average As Double, _
    ByRef dutyCycle As Double, _
    ByRef pf As PASS_FAIL_RESULT) _
    As Long
```

VB.NET

```
Public Declare Function LB_MeasurePulse_PF _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef pulse As Double, _
    ByRef peak As Double, _
    ByRef average As Double, _
    ByRef dutyCycle As Double, _
    ByRef pf As PASS_FAIL_RESULT) _
    As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_MeasurePulse_PF(int addr,
    ref double pulse,
    ref double peak,
    ref double average,
    ref double dutyCycle,
    ref PASS_FAIL_RESULT pf );
```

LB_ResetRegStates (and related commands)

Related Commands:

LB_ResetCurrentState

These commands allow the user to cause either the current state or the state information held in the save/recall registers to be reset.

Pass Parameters:

addr – address of the device

Returned Values:

Success: >=1

Failure: < 0

Command Group:

Save/Recall

(See page 18, *Save/Recall Command Group*.)

Sample Code Declarations:

C++	long LB_ResetCurrentState(long addr);
	long LB_ResetRegStates(long addr);

VB.NET	Public Declare Function LB_ResetCurrentState Lib "LB_API2.dll" (ByVal addr As Integer) As Integer Public Declare Function LB_ResetRegStates Lib "LB_API2.dll" (ByVal addr As Integer) As Integer

C#	System.Runtime.InteropServices.DllImport("LB_API2.dll")]
	Int LB_ResetCurrentState(int addr);
	[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
	Int LB_ResetRegStates(int addr);

LB_ReadStateFromINI (and related commands)

Related Commands:

LB_WriteStateToINI

These commands cause the current state, including all numbered registers to be written to an INI file. The resulting INI file is located in the directory of execution. The name of the INI file will be the model number concatenated with the serial number separated by and underscore. For example, a PSM4410 with a serial number of 074102 would result in an file named PSM4410_074102.INI. The parameters and values are written in text form so they are human-readable.

Pass Parameters:

addr – address of the device

Returned Values:

Success: >=1

Failure: < 0

Command Group:

Save/Recall

(See page 18, *Save/Recall Command Group*.)

Sample Code Declarations:

C++

```
long LB_ReadStateFromINI(long addr);
long LB_WriteStateToINI(long addr);
```

VB.NET

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

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_ReadStateFromINI(int addr);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_WriteStateToINI(int addr);
```

LB_SensorCnt

This command returns the number of instruments currently connected to the computer.

Pass Parameters:

None

Returned Values:

Success: The number of instruments connected to the PC. The number will be between 0 and 16

Failure: Any number < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++	LB_API2 long _stdcall LB_SensorCnt();
VB 6.0	Public Declare Function LB_SensorCnt Lib "LB_API2.dll" () As Long
VB.NET	Public Declare Function LB_SensorCnt Lib "LB_API2.dll" () As Integer
C#	[System.Runtime.InteropServices.DllImport("LB_API2.dll")] public static extern int LB_SensorCnt();

LB_SensorList

This command returns a description for each instrument. The user must ensure that an array of instrument descriptions have been properly allocated. The number of descriptions returned will be equivalent to the number returned in LB_SensorCnt.

Note the differences in the declaration of the structures. Converting the byte data to a more sensible structure is demonstrated in the address management utilities.

Pass Parameters:

A properly sized array of instrument descriptions.

Returned Values:

Success: > 0, 1 plus the number if items

Failure:< 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
struct SensorDescrption
{
    long DeviceIndex;       // 1..n
    long DeviceAddress;     // 1..255
    char SerialNumber[7];   // zero terminated 6 char string
};

LB_API2
long _stdcall LB_
    SensorList( SensorDescrption* SD,
    long cnt)
```

VB 6.0

```
Public Type SDByte
    DeviceIndex As Long
    DeviceAddress As Long
    SerialNumber(0 To 6) As Byte
End Type

Public Declare Function LB_SensorList Lib "LB_API2.dll" ( _
                                ByRef SD As SDByte, _
                                ByVal cnt As Long) _
                                 As Long
```

VB.NET

```
Public Structure SDByte
    Dim DeviceIndex As Integer
    Dim DeviceAddress As Integer
    Dim SNByte0, _
        SNByte1, _
        SNByte2, _
        SNByte3, _
        SNByte4, _
        SNByte5, _
        SNByte6 As Byte
End Structure

Public Declare Function LB_SensorList Lib "LB_API2.dll" ( _
                        ByRef sd As SDByte, _
                        ByVal cnt As Integer) _
                         As Integer
```

C#

```
public struct SDByte
 {
    public int DeviceIndex;
    public int DeviceAddress;
   public byte SNByte0, SNByte1, SNByte2, SNByte3, SNByte4, SNByte5, SNByte6;
 }

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SensorList(
    ref SDByte sd,
    int cnt );
```

LB_Set75OhmsEnabled (and related commands)

Related Commands:

LB_Get75OhmsEnabled

These commands are used to correct measurements made using a minimum loss pad. Minimum loss pads (sometimes referred to as L-pad attenuators) are used to match 75 ohm systems to 50 ohm systems. The correction is about 5.72 dB.

Pass Parameters:

addr – address of the device

st – indicates the state of extended averaging, 0 = off, 1 = on,

Returned Values:

Success: >=1

Failure: < 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
long LB_Get75OhmsEnabled(long addr, enum FEATURE_STATE* st);
long LB_Set75OhmsEnabled(long addr, enum FEATURE_STATE st);
```

VB.NET

```
Public Declare Function LB_Get75OhmsEnabled Lib "LB_API2.dll"
        (ByVal addr As Integer,
       ByRef st As FEATURE_STATE) As Integer

Public Declare Function LB_Set75OhmsEnabled Lib "LB_API2.dll"
        (ByVal addr As Integer,
        ByVal st As FEATURE_STATE) As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
Int LB_Get75OhmsEnabled(int addr, ref FEATURE_STATE st);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
Int LB_Set75OhmsEnabled(int addr, FEATURE_STATE st);
```

LB_SetAddress_Idx (and related commands)

Related Commands:

LB_GetAddress_Idx

These commands return the address, given the index and vice versa. The index is assigned by the OS when the unit is plugged in.

Pass Parameters:

The index is passed, which will normally be between 1 and 16. In LB_SetAddress_Idx, the address is also passed and valid values are between 1 and 255.

Returned Values:

Success: > The address between 1 and 255 for getting the address and >0 for setting the address

Failure: < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_GetAddress_Idx(long idx);

LB_API2 long _stdcall LB_SetAddress_Idx(long idx, long addr);
```

VB 6.0

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

Public Declare Function LB_SetAddress_Idx _
                          Lib " LB_API2.dll" ( _
                          ByVal idx As Long, _
                          ByVal addr As Long) _
                           As Long
```

VB.NET

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

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

C#

```
[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_SetAddress_Idx( int idx, int addr );
```

LB_SetAddress_SN (and related commands)

Related Commands:

LB_GetAddress_SN

These commands return the address, given the serial number and vice versa.

Pass Parameters:

The serial number is passed in both cases. It should be six characters in length, plus one character for the zero termination.

In LB_SetAddress_SN, the address is also passed.

Returned Values:

Success: > The address between 1 and 255

Failure: < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_GetAddress_SN
       char* SN);

LB_API2 long _stdcall LB_SetAddress_SN(
       char* SN,
       long addr);
```

VB 6.0

```
Public Declare Function LB_GetAddress_SN _
                                       Lib "LB_API2.dll" ( _
                                       ByVal sn As String) _
                                        As Long

Public Declare Function LB_SetAddress_SN _
                                       Lib "LB_API2.dll" ( _
                                       ByVal sn As String, _
                                       ByVal addr As Long) _
                                        As Long
```

VB.NET

```
Public Declare Function LB_GetAddress_SN _
                            Lib "LB_API2.dll" ( _
                            ByVal sn As String) _
                             As Integer

Public Declare Function LB_SetAddress_SN _
                            Lib "LB_API2.dll" ( _
                            ByVal sn As String, _
                            ByVal addr As Integer) _
                             As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetAddress_SN(
    string sn );

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

LB_SetAntiAliasingEnabled (and related commands)

Related Commands:

LB_GetAntiAliasingEnabled

These commands enable or disable the anti-aliasing feature or allow its state to be checked.

Normally, the sampling rate is 500 kHz. As the baseband signals approach the Nyquist criteria (realistically about 200 kHz in this case) problems arise. These are addressed using an anti-aliasing capability that in effect randomizes the samples. This randomization does have some affect on the rapidity of acquiring the data.

As a result, the anti-aliasing algorithm is normally turned off. However, when measuring signals that have baseband content greater than about 200 kHz, turning on the anti-aliasing feature is recommended.

Pass Parameters:

Address, feature state (1 = on, 0 = off)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetAntiAliasingEnabled(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_GetAntiAliasingEnabled(
      long addr,
      FEATURE_STATE* st);
```

VB 6.0

```
Public Declare Function LB_SetAntiAliasingEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal st As FEATURE_STATE) _
      As Long

Public Declare Function LB_GetAntiAliasingEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByRef st As FEATURE_STATE) _
      As Long
```

VB.NET

```
Public Declare Function LB_SetAntiAliasingEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByVal st As FEATURE_STATE) _
       As Integer

Public Declare Function LB_GetAntiAliasingEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef st As FEATURE_STATE) _
       As Integer
```

C#

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

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetAntiAliasingEnabled(
       int addr,
       ref FEATURE_STATE st );
```

LB_SetAutoPulseEnabled (and related commands)

Related Commands:

LB_GetAutoPulseEnabled

These commands enable or disable the default or automatic pulse measurement criteria. The default value is 3 dB below the measured peak value. This means that when this feature is enabled, the pulse power will be the average of all power greater than 3 dB below peak.

For example, if the peak was measured to be -30 dBm and this feature was enabled, all samples greater than -33dBm would be included as pulse power. If this criteria is disabled, then the value set using LB_GetPulseCriteria would be used. Additional functions that may be of interest are:

- LB_GetPulseCriteria
- LB_SetPulseCriteria

Pass Parameters:

Address, state of the feature (1 = on, 0 = off)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Pulse Setup

(See page 12, *Pulse Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetAutoPulseEnabled(
        long addr,
        FEATURE_STATE st);

LB_API2 long _stdcall LB_GetAutoPulseEnabled(
        long addr,
        FEATURE_STATE* st);
```

VB 6.0

```
Public Declare Function LB_SetAutoPulseEnabled _
        Lib "LB_API2.dll" ( _
        ByVal addr As Long, _
        ByVal state As FEATURE_STATE) _
        As Long

Public Declare Function LB_GetAutoPulseEnabled _
        Lib "LB_API2.dll" ( _
        ByVal addr As Long, _
        ByRef state As FEATURE_STATE) _
        As Long
```

VB.NET

```
Public Declare Function LB_SetAutoPulseEnabled _
        Lib "LB_API2.dll" ( _
        ByVal addr As Integer, _
        ByVal state As FEATURE_STATE) _
        As Integer

Public Declare Function LB_GetAutoPulseEnabled _
        Lib "LB_API2.dll" ( _
        ByVal addr As Integer, _
        ByRef state As FEATURE_STATE) _
        As Integer
```

C#

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

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetAutoPulseEnabled(
        int addr,
        ref FEATURE_STATE state );
```

LB_SetAverages (and related commands)

Related Commands:

LB_GetAverages

These commands set or get the number of data buffers that are averaged. The default is set to 75. It typically takes about 0.3 to 1 ms to collect one buffer of data.

Pass Parameters:

Address, averages (1 to 30000)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++	LB_API2 long _stdcall LB_SetAverages(long addr, long value); LB_API2 long _stdcall LB_GetAverages(long addr, long* averages);
VB 6.0	Public Declare Function LB_GetAverages _ Lib "LB_API2.dll" (_ ByVal addr As Long, _ ByRef value As Long) _ As Long Public Declare Function LB_SetAverages _ Lib "LB_API2.dll" (_ ByVal addr As Long, _ ByVal value As Long) _ As Long

VB.NET

```
Public Declare Function LB_GetAverages _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef value As Integer) _
       As Integer

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

C#

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

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

LB_SetCalDueDate (and related commands)

Related Commands:

LB_GetCalDueDate

These commands set or get the calibration due date, which is specified by serial number, not address. The calibration due date is set at the factory, but can be changed as an instrument is calibrated.

Pass Parameters:

NA

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Service

(See page 19, *Service Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetCalDueDate(
        char* SN,
        long lngYear,
        long lngMonth,
        long lngDay);

LB_API2 long _stdcall LB_GetCalDueDate(
        char* SN,
        long* year,
        long* month,
        long* day);
```

VB 6.0

```
Public Declare Function LB_SetCalDueDate _
        Lib "LB_API2.dll" ( _
        ByVal sn As String, _
        ByVal year As Long, _
        ByVal month As Long, _
        ByVal day As Long) _
        As Long

Public Declare Function LB_GetCalDueDate _
        Lib "LB_API2.dll" ( _
        ByVal sn As String, _
        ByRef day As Long, _
        ByRef month As Long, _
        ByRef day As Long) _
        As Long
```

VB.NET

```
Public Declare Function LB_SetCalDueDate _
      Lib "LB_API2.dll" ( _
      ByVal sn As String, _
      ByVal year As Integer, _
      ByVal month As Integer, _
      ByVal day As Integer) _
      As Integer

Public Declare Function LB_GetCalDueDate _
      Lib "LB_API2.dll" ( _
      ByVal sn As String, _
      ByRef day As Integer, _
      ByRef month As Integer, _
      ByRef day As Integer) _
      As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetCalDueDate(
      string sn,
      int year,
      int month,
      int day );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetCalDueDate(
      string sn,
      ref int year,
      ref int month,
      ref int day);
```

LB_SetCWReference (and related commands)

Related Commands:

LB_GetCWReference

These commands set up the instrument for relative measurements during CW measurements. (Separate functions set the reference for pulse measurements.) To make relative measurements, the units of measure must be set to "dB Relative".

For more information, see the functions LB_SetMeasurementPowerUnits and LB_GetMeasurementPowerUnits. The reference may be changed during a relative measurement. All relative measurements are made as a ratio and reported as dB above or below the reference.

Other commands of interest:

LB_SetMeasurementPowerUnits

LB_GetMeasurementPowerUnits

Pass Parameters:

Address, reference level, power units. The units enumeration is shown below:

DBM = 0	' dBm
DBW = 1	' dBW
DBKW = 2	' dBkW
DBUV = 3	' dBuV
W = 4	' Watts
V = 5	' Volts
DBREL = 6	' dB Relative **(**INVALID FOR SETTING A REFERENCE)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetCWReference(long addr, double relRef, PWR_UNITS units);
LB_API2 long _stdcall LB_GetCWReference(long addr, double* relRef, PWR_UNITS* units);
```

VB 6.0

```
Public Declare Function LB_SetCWReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal Ref As Double, _
    ByVal units As PWR_UNITS) _
    As Long

Public Declare Function LB_GetCWReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef relRef As Double, _
    ByRef units As PWR_UNITS) _
    As Long
```

VB.NET

```
Public Declare Function LB_SetCWReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal Ref As Double, _
    ByVal units As PWR_UNITS) _
    As Integer

Public Declare Function LB_GetCWReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef relRef As Double, _
    ByRef units As PWR_UNITS) _
    As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetCWReference(
    int addr,
    double Ref,
    PWR_UNITS units );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetCWReference(
    nt addr,
    ref double relRef,
    ref PWR_UNITS units );
```

LB_SetDutyCycleEnabled (and related commands)

Related Commands:

LB_GetDutyCycleEnabled

LB_SetDutyCyclePerCent

LB_GetDutyCyclePerCent

These commands set up the instrument for average pulse power measurements based on an assumed duty cycle value. These commands may be used on the PSM3000 to measure average pulse power. This technique is not recommended for the PSM4000 or PSM5000 Series, even thought the command is compatible with them. Direct average pulse power measurements are available on the PSM4000 and PSM5000 through the LB_MeasurePulse command.

The calculation to adjust for duty cycle is:

$10Log_{10}$(Duty Cycle)

Assuming a duty cycle of 10%, the calculation for equivalent average power would be:

$10Log_{10}(0.1) = -10$ dB

This means the average power of a signal with a 10% duty cycle will be 10 dB below the peak value. For instruments measuring average power, the power reading is simply adjusted by 10 dB. This adjustment yields the peak power but it also assumes that the duty cycle is correct.

These commands can enable or disable the duty cycle feature, and set the percent value. SetDutyCycleEnabled enables or disables the duty cycle adjustment. GetDutyCycleEnabled reads back the state of this feature (enabled or disabled). SetDutyCyclePerCent sets the value of the duty cycle without affecting the state of the feature. GetDutyCyclePerCent reads back the value of the duty cycle.

Pass Parameters:

Address, feature state (1 = on, 0 = off) or the value of the duty cycle in percent.

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetDutyCycleEnabled(
       long addr,
       FEATURE_STATE st);

LB_API2 long _stdcall LB_GetDutyCycleEnabled(
       long addr,
       FEATURE_STATE* st);

LB_API2 long _stdcall LB_SetDutyCyclePerCent(
       long addr,
       double val);

LB_API2 long _stdcall LB_GetDutyCyclePerCent(
       long addr,
       double* val);
```

VB 6.0

```
Public Declare Function LB_SetDutyCycleEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByVal state As FEATURE_STATE) _
       As Long

Public Declare Function LB_GetDutyCycleEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByRef state As FEATURE_STATE) _
       As Long

Public Declare Function LB_SetDutyCyclePerCent _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByVal val As Double) _
       As Long

Public Declare Function LB_GetDutyCyclePerCent _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByRef val As Double) _
       As Long
```

VB.NET

```
Public Declare Function LB_SetDutyCycleEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal state As FEATURE_STATE) _
    As Integer

Public Declare Function LB_GetDutyCycleEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef state As FEATURE_STATE) _
    As Integer

Public Declare Function LB_SetDutyCyclePerCent _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal val As Double) _
    As Integer

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

C#

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

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

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

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

LB_SetExtendedAveraging (and related commands)

Related Commands:

LB_GetExtendedAveraging

LB_SetExtendedAveragingEnabled

LB_GetExtendedAveragingEnabled

LB_ResetExtendedAveraging

These commands manage the extended average settings of the instrument. Regular averaging performs a series of acquisitions and then delivers a final measurement. Extended averaging "extends" over multiple measurements. Extended averaging uses a form of exponential averaging that combines the most recent measurement with previous measurements. Regular averaging and extended averaging may be used together to achieve good stability and good response time. All averaging is performed on linear data regardless of the current measurement units.

In programmatic measurements, it is important to get a good measurement as quickly as possible. Using extended averaging allows for long measurement times, but allows you to stop the measurement when a stable reading has been achieved. The pseudo-code below shows how that might be achieved:

```
LB_SetExtendedAveraging(50)                 'sets the number of extended averages to 50
LB_SetAverages (400)                        'sets averaging to about 100 µs
LB_SetExtendedAveragingEnabled(true)        'enables extended averaging
LB_ResetExtendedAveraging ()                'resets (restarts) extending averaging process
LastMeas = -1000
DeltaMeas = -1000
DeltaMeasLimit = 0.04                       'causes the loop to exit when the change in the
                                            'last two measurements is less than 0.04 dB
Do
    LB_MeaseureCW(cw)
    DeltaMeas = AbsoluteValue(cw – LastMeas)    'make a measurment
    LastMeas = cw                           'calculate the change
While (DeltaMeas > DeltaMeasLimit)          'retain the last measurement
FinalResult = LastMeas                      'loop until
LB_ResetExtendedAveraging ()
LB_SetExtendedAveragingEnabled(false)
```

This routine makes measurements until the result becomes stable with the result returned in the variable "FinalResult". Yet it measures for no more than an extra 100µs. In actual use, additional code would have to be added for error checking and trapping.

Extended averaging remembers the essential elements of recent measurements. Each call to LB_ResetExtendedAveraging clears all the data held in the buffers and restarts the averaging process. Each subsequent measurement add more averaging until the number of measurements set by calling LB_SetExtendedAveraging has been made. Then the average

The commands function as follows:

- LB_GetExtendedAveragingEnabled – returns the state of extended averaging.
- LB_SetExtendedAveragingEnabled – enables (st = 1) or disables (st = 0) extended averaging.
- LB_GetExtendedAveraging – returns the number of extended averages.

- LB_SetExtendedAveraging – sets the number of extended averages. The minimum is 1 the maximum is limited by the the type of variable.
- LB_ResetExtendedAveraging – clearly all the buffers related to extended averaging and restarts the process.

Pass Parameters:

addr – address of the device

st – indicates the state of extended averaging, 0 = off, 1 = on,

extAvg – number of most recent measurements to include in the average

Returned Values:

Success: >= 1

Failure: < 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
long LB_GetExtendedAveragingEnabled(long addr, enum FEATURE_STATE* st);
long LB_SetExtendedAveragingEnabled(long addr, enum FEATURE_STATE st);
long LB_GetExtendedAveraging(long addr, long* extAvg);
long LB_SetExtendedAveraging(long addr, long extAvg);
long LB_ResetExtendedAveraging(long addr);
```

VB.NET

```
Public Declare Function LB_GetExtendedAveragingEnabled Lib "LB_API2.dll" (ByVal addr As Integer, ByRef st As FEATURE_STATE) As Integer

Public Declare Function LB_SetExtendedAveragingEnabled Lib "LB_API2.dll" (ByVal addr As Integer, ByVal st As FEATURE_STATE) As Integer

Public Declare Function LB_GetExtendedAveraging Lib "LB_API2.dll" (ByVal addr As Integer, ByRef extAvg As Integer) As Integer

Public Declare Function LB_SetExtendedAveraging Lib "LB_API2.dll" (ByVal addr As Integer, ByVal extAvg As Integer) As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_GetExtendedAveragingEnabled(int addr, ref FEATURE_STATE st);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_SetExtendedAveragingEnabled(int addr, FEATURE_STATE st);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_GetExtendedAveraging(int addr, ref int extAvg);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_SetExtendedAveraging(int addr, int extAvg);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_ResetExtendedAveraging(int addr);
```

LB_SetFrequency (and related commands)

Related Commands:

LB_GetFrequency

These commands set or get the frequency of the addressed device. Frequency is specified in Hz. It is important to note the necessity of setting the frequency to get accurate measurements.

Pass Parameters:

Address, frequency in Hz.

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetFrequency(long addr, double value);
LB_API2 long _stdcall LB_GetFrequency(long addr, double* value);
```

VB 6.0

```
Public Declare Function LB_SetFrequency _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal value As Double) _
    As Long

Public Declare Function LB_GetFrequency _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef value As Double) _
    As Long
```

VB.NET

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

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

C#

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

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

LB_SetLimitEnabled (and related commands)

Related Commands:

LB_GetLimitEnabled

LB_SetSingleSidedLimit

LB_GetSingleSidedLimit

LB_SetDoubleSidedLimit

LB_GetDoubleSidedLimit

These commands set and get limits, and specify single-sided limits, double- sided limits, or neither limit.

Limits are fixed values against which a measured value is compared and typically evaluated as pass or fail. This evaluation is made and returned during either LB_MeasureCW_PF or LB_MeasurePulse_PF.

There are two types of limits:

- Single line — the value can be below, equal to, or above this line. Any of these conditions can be specified as pass or fail. Typicallly, the passing condition is specified; failing is implied by not passing.
- Double line — the value can be equal outside these lines, between the lines, or equal to one of the lines. Any condition may be specified as pass or fail.

The following is required when specifying a limit:

- Address (instrument to which this applies)
- Type of limit
- Boundary conditions (one for single-sided, two for double-sided)
- Units for the boundary conditions
- The rule of how to evaluate a pass or fail (we specify pass). The rules are different for single and double-sided limits.

Enabling the units means specifying to enable single-sided limits, double-sided limits, or neither limit.

NOTE. *In the declarations section below, some of the comments have been truncated for brevity. Also, the enumerations specific to limits are shown, with the exception of units, which are shown in several other areas.*

Pass Parameters:

Address, value(s) of the limit(s), units and the rule.

OR

Address, feature state (1 = on, 0 = off)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
enum LIMIT_STYLE                        // Enumeration of pass/fail limits
{
    LIMITS_OFF = 0,
    SINGLE_SIDED = 1,
    DOUBLE_SIDED = 2
};

enum SS_RULE
{
    PASS_LT = 0,                        // Pass if measured value less than
    PASS_LTE = 1,                       // Pass if measured value less than or equal
    PASS_GT = 2,                        // Pass if measured value greater than
    PASS_GTE = 3,                       // Pass if measured value greater than or equal
};


enum DS_RULE
{
    PASS_BETWEEN_EXC = 0,               // Pass if value is greater than lower limit AND less than upper
                                        limit
    PASS_BETWEEN_INC = 1,               // Pass if value is equal to or greater than lower limit AND equal to
                                        // or less than upper limit
    PASS_OUTSIDE_EXC = 2,               // Pass if measured value is less than the lower OR
                                        // greater than the upper limit
    PASS_OUTSIDE_INC = 3                // Pass if measured value is equal to or greater than the upper
                                        // limit OR equal to or less than the lower limit

enum PASS_FAIL_RESULT
{
    PASS = 0,                           // pass, measured value within limits
    FAIL_LOW = 1,                       // failed, measured value too low
    FAIL_HIGH = 2,                      // failed, measured value too high
    FAIL_BETWEEN_LIMIT_EXC = 3,         // failed greater than or equal to lower limit
    FAIL_BETWEEN_LIMIT_INC = 4,         // failed less than or equal to lower limit
    NO_DETERMINATION = 5                // no determination made, possible reasons include
};                                      // the following reasons:
                                        // - limits are not enabled
                                        // - limits are not specified
                                        // - valid measurement not made (timeout?)
```

```
LB_API2 long _stdcall
LB_SetLimitEnabled(
        long addr,
        LIMIT_STYLE lmtStyle);

LB_API2 long _stdcall
LB_SetSingleSidedLimit(
        long addr,
        double val,
        PWR_UNITS units,
        SS_RULE passFail);

LB_API2 long _stdcall
LB_SetDoubleSidedLimit(
        long addr,
        double lowerVal,
        double upperVal,
        PWR_UNITS units,
        DS_RULE passFail);

LB_API2 long _stdcall
LB_GetLimitEnabled(
        long addr,
        LIMIT_STYLE* lmtStyle);

LB_API2 long _stdcall
LB_GetSingleSidedLimit(
        long addr,
        double* val,
        PWR_UNITS* units,
        SS_RULE* passFail);

LB_API2 long _stdcall
LB_GetDoubleSidedLimit(
        long addr,
        double* lowerVal,
        double* upperVal,
        PWR_UNITS* units,
        DS_RULE* passFail);
```

VB 6.0

```
Public Enum LIMIT_STYLE
      LIMITS_OFF = 0                        ' Disable limits
      SINGLE_SIDED = 1                      ' Use single sided limits in pass/fail evaluation
      DOUBLE_SIDED = 2                      ' Use double sided limits in pass/fail evaluation
End Enum

Public Enum SS_RULE
      PASS_LT = 0                           ' Pass if measured value less than
      PASS_LTE = 1                          ' Pass if measured value less than or equal
      PASS_GT = 2                           ' Pass if measured value greater than
      PASS_GTE = 3                          ' Pass if measured value greater than or equal
End Enum

Public Enum DS_RULE
      PASS_BETWEEN_EXC = 0                  ' Pass if measured value is greater than the
      PASS_BETWEEN_INC = 1                  ' Pass if measured value is equal to or greater
      PASS_OUTSIDE_EXC = 2                  ' Pass if measured value is less than the lower
      PASS_OUTSIDE_INC = 3                  ' Pass if measured value is equal to or greater
End Enum

Public Enum PASS_FAIL_RESULT
      PASS = 0                              ' Pass measured value within limits
      FAIL_LOW = 1                          ' Failed measured value too low
      FAIL_HIGH = 2                         ' Failed measured value too high
      FAIL_BETWEEN_LIMIT_EXC = 3            ' Failed between limits
      FAIL_BETWEEN_LIMIT_INC = 4            ' Failed between limits
      NO_DETERMINATION = 5reasons           ' No determination made; possible
End Enum                                    ' reasons include:
Public  Declare  Function                   ' - limits are not enabled
LB_SetLimitEnabled _                        ' - limits are not specified
      Lib "LB_API2.dll" ( _                 ' - valid measurement not made (timeout?)
      ByVal addr As Long, _
      ByVal lmtStyle As LIMIT_STYLE) _
      As Long
```

```
Public Declare Function
LB_SetSingleSidedLimit _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal val As Double, _
    ByVal units As PWR_UNITS, _
    ByVal passFail As SS_RULE) _
    As Long
Public Declare Function
LB_SetDoubleSidedLimit _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal lowerVal As Double, _
    ByVal upperVal As Double, _
    ByVal units As PWR_UNITS, _
    ByVal passFail As DS_RULE) _
    As Long
Public Declare Function
LB_GetLimitEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef lmtStyle As LIMIT_STYLE) _
    As Long
Public Declare Function
LB_GetSingleSidedLimit _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef val As Double, _
    ByRef units As PWR_UNITS, _
    ByRef passFail As SS_RULE) _
    As Long
Public Declare Function
LB_GetDoubleSidedLimit _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef lowerVal As Double, _
    ByRef upperVal As Double, _
    ByRef units As PWR_UNITS, _
    ByRef passFail As DS_RULE) _
    As Long
```

VB.NET

```
Public Enum LIMIT_STYLE
      LIMITS_OFF = 0                    'disable limits
      SINGLE_SIDED = 1                  'use single sided limits in pass/fail evaluation
      DOUBLE_SIDED = 2                  'use double sided limits in pass/fail evaluation
End Enum

Public Enum SS_RULE
      PASS_LT = 0                       ' Pass if measured value less than
      PASS_LTE = 1                      ' Pass if measured value less than or equal
      PASS_GT = 2                       ' Pass if measured value greater than
      PASS_GTE = 3                      ' Pass if measured value greater than or equal
End Enum

Public Enum DS_RULE
      PASS_BETWEEN_EXC = 0              ' Pass if measured value is greater than the
      PASS_BETWEEN_INC = 1              ' Pass if measured value is equal to or greater
      PASS_OUTSIDE_EXC = 2              ' Pass if measured value is less than the lower
      PASS_OUTSIDE_INC = 3              ' Pass if measured value is equal to or greater
End Enum

Public Enum PASS_FAIL_RESULT
      PASS = 0                          ' pass measured value within limits
      FAIL_LOW = 1                      ' failed measured value too low
      FAIL_HIGH = 2                     ' failed measured value too high
      FAIL_BETWEEN_LIMIT_EXC = 3        ' failed between limits
      FAIL_BETWEEN_LIMIT_INC = 4        ' failed between limits
      NO_DETERMINATION = 5 reasons      ' no determination made; possible
                                        ' reasons include:
                                        ' - limits are not enabled
                                        ' - limits are not specified
                                        ' - valid measurement not made (timeout?)
End Enum

Public  Declare  Function
LB_SetLimitEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal lmtStyle As LIMIT_STYLE) _
      As Integer

Public  Declare  Function
LB_SetSingleSidedLimit _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal val As Double, _
      ByVal units As PWR_UNITS, _
      ByVal passFail As SS_RULE) _
      As Integer
```

```
Public  Declare  Function
LB_SetDoubleSidedLimit _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByVal lowerVal As Double, _
       ByVal upperVal As Double, _
       ByVal units As PWR_UNITS, _
       ByVal passFail As DS_RULE) _
        As Integer

Public  Declare  Function
LB_GetLimitEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef lmtStyle As LIMIT_STYLE) _
        As Integer

Public  Declare  Function
LB_GetSingleSidedLimit _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef val As Double, _
       ByRef units As PWR_UNITS, _
       ByRef passFail As SS_RULE) _
        As Integer

Public  Declare  Function
LB_GetDoubleSidedLimit _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef lowerVal As Double, _
       ByRef upperVal As Double, _
       ByRef units As PWR_UNITS, _
       ByRef passFail As DS_RULE) _
        As Integer
```

C#

```
public enum LIMIT_STYLE
{
       LIMITS_OFF = 0,
       SINGLE_SIDED = 1,
       DOUBLE_SIDED = 2,
}

public enum SS_RULE
       PASS_LT = 0,
       PASS_LTE = 1,
       PASS_GT = 2,
       PASS_GTE = 3,
}

public enum DS_RULE
{
       PASS_BETWEEN_EXC = 0,
       PASS_BETWEEN_INC = 1,
       PASS_OUTSIDE_EXC = 2,
       PASS_OUTSIDE_INC = 3,
}
```

```
public enum PASS_FAIL_RESULT
{
       PASS = 0,
       FAIL_LOW = 1,
       FAIL_HIGH = 2,
      FAIL_BETWEEN_LIMIT_EXC = 3,
      FAIL_BETWEEN_LIMIT_INC = 4,
      NO_DETERMINATION = 5,
     // not limited to the following reasons:
      // - limits are not enabled
      // - limits are not specified
       // - valid measurement not made
(timeout?)
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_SetLimitEnabled(
       int addr,
      LIMIT_STYLE lmtStyle );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_SetSingleSidedLimit(
       int addr,
       double val,
      PWR_UNITS units,
      SS_RULE passFail );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_SetDoubleSidedLimit(
       int addr,
      double lowerVal,
      double upperVal,
      PWR_UNITS units,
      DS_RULE passFail );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_GetLimitEnabled(
       int addr,
      ref LIMIT_STYLE lmtStyle );
```

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_GetSingleSidedLimit(
       int addr,
       ref double val,
       ref PWR_UNITS units,
       ref SS_RULE passFail );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int
LB_GetDoubleSidedLimit(
       int addr,
       ref double lowerVal,
       ref double upperVal,
       ref PWR_UNITS units,
       ref DS_RULE passFail );
```

LB_SetMaxHoldEnabled (and related commands)

Related Commands:

LB_GetMaxHoldEnabled

LB_ResetMaxHold

These commands cause CW or pulse measurements to retain the greater of the most recent measurement or previous measurements. Resetting max hold restarts the process of looking for a new maximum. In CW only the CW or average power measurement is affected by max hold. In pulse measurements all values (pulse, peak, duty cycle and average) are affected by max hold.

LB_GetMaxHoldEnabled returns the state of the max hold feature. LB_SetMaxHoldEnabled disables (0) or enables (1) the max hold feature. Finally, LB_ResetMaxHold restarts the search for a maximum.

Pass Parameters:

addr – address of the device

st – indicates the state of extended averaging, 0 = off, 1 = on,

Returned Values:

Success: >=1

Error: <0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
long LB_GetMaxHoldEnabled(long addr, enum FEATURE_STATE* st);
long LB_SetMaxHoldEnabled(long addr, enum FEATURE_STATE st);
long LB_ResetMaxHold(long addr);
```

VB.NET

```
Public Declare Function LB_GetMaxHoldEnabled Lib "LB_API2.dll"
        (ByVal addr As Integer,
       ByRef st As FEATURE_STATE) As Integer

Public Declare Function LB_SetMaxHoldEnabled Lib "LB_API2.dll"
        (ByVal addr As Integer,
       ByVal st As FEATURE_STATE) As Integer

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

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_GetMaxHoldEnabled(int addr, ref FEATURE_STATE st);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_SetMaxHoldEnabled(int addr, enum FEATURE_STATE st);

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int LB_ResetMaxHold(int addr);
```

LB_SetMeasurementPowerUnits (and related commands)

Related Commands:

LB_GetMeasurementPowerUnits

These commands set or get the measurement power units. The available units are:

DBM = 0	' dBm
DBW = 1	' dBW
DBKW = 20	' dBkW
DBUV = 3	' dBuV
W = 4	' Watts
V = 5	' Volts
DBREL = 6	' dB Relative

When the units are set to "DBREL", the measurement is always in dB. When the instrument is measuring CW, the CW reference is used. When making pulse measurements, the pulse reference values are used as the basis for the measurements.

Other commands of interest:

LB_SetCWReference

LB_GetCWReference

LB_SetPulseReference

LB_GetPulseReference

Pass Parameters:

Address, power units. The units enumeration is shown above:

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetMeasurementPowerUnits(
      long addr,
      PWR_UNITS units);

LB_API2 long _stdcall LB_GetMeasurementPowerUnits(
      long addr,
      PWR_UNITS* units);
```

VB 6.0

```
Public Declare Function LB_SetMeasurementPowerUnits _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal units As PWR_UNITS) _
      As Long

Public Declare Function LB_GetMeasurementPowerUnits _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByRef units As PWR_UNITS) _
      As Long
```

VB.NET

```
Public Declare Function LB_SetMeasurementPowerUnits _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal units As PWR_UNITS) _
      As Integer

Public Declare Function LB_GetMeasurementPowerUnits _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByRef units As PWR_UNITS) _
      As Integer
```

C#

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

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetMeasurementPowerUnits(
      int addr,
      ref PWR_UNITS units );
```

LB_SetOffset (and related commands)

Related Commands:

LB_GetOffset

LB_SetOffsetEnabled

LB_GetOffsetEnabled

These commands cause a fixed offset to be added to the reading, or enable/disable the feature. The offset is typically used to compensate for losses or gains in the measurement path. This offset is fixed and is not a function of frequency. For an offset that is a function of frequency, use the response function calls.

Pass Parameters:

Address, and either the state of the feature (1=on, 0 = off) or the value of the offset.

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetOffsetEnabled(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_GetOffsetEnabled(
      long addr,
      FEATURE_STATE* st);

LB_API2 long _stdcall LB_SetOffset(
      long addr,
      double val);

LB_API2 long _stdcall LB_GetOffset(
      long addr,
      double* val);
```

VB 6.0

```
Public Declare Function LB_SetOffsetEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal state As FEATURE_STATE) _
    As Long

Public Declare Function LB_GetOffsetEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef state As FEATURE_STATE) _
    As Long

Public Declare Function LB_SetOffset _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal val As Double) _
    As Long

Public Declare Function LB_GetOffset _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef val As Double) _
    As Long
```

VB.NET

```
Public Declare Function LB_SetOffsetEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal state As FEATURE_STATE) _
    As Integer

Public Declare Function LB_GetOffsetEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef state As FEATURE_STATE) _
    As Integer

Public Declare Function LB_SetOffset _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal val As Double) _
    As Integer

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

C#

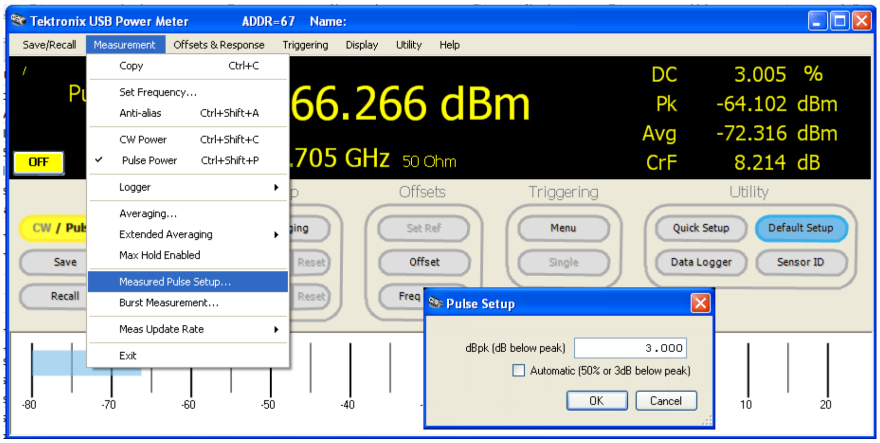
```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetOffsetEnabled(
    int addr,
    FEATURE_STATE state );

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

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

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

LB_SetPulseCriteria (and related commands)

Related Commands:

LB_GetPulseCriteria

These commands set or get the pulse measurement criteria. This value determines what portion of the pulse will be used to measure pulse power. The default or automatic value is 3 dB below the measured peak value, or the 50% down points. You can also set the criteria and leave the automatic feature on. Then, by turning the auto feature on and off, you can switch between the automatic value (dB) and the desired value . For instance, if the criteria is set to 6dB and the auto criteria is turned on and off, you can toggle between 3 and 6 dB below peak. This can also provide some sense of rise time or slope, and the sensitivity to this criteria.



66.266 dBm

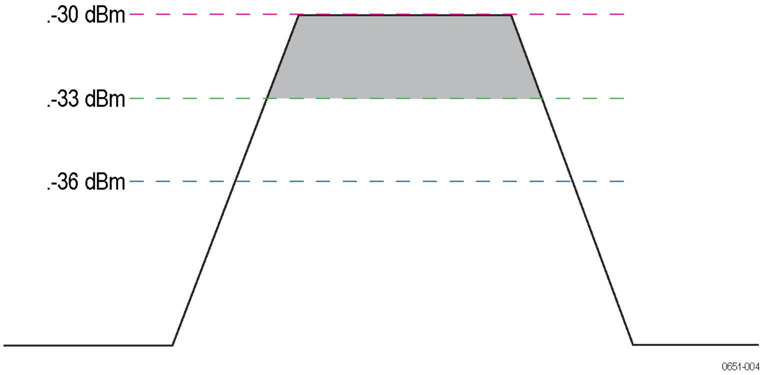
.705 GHz 50 Ohm

DC 3.005 %
Pk -64.102 dBm
Avg -72.316 dBm
CrF 8.214 dB

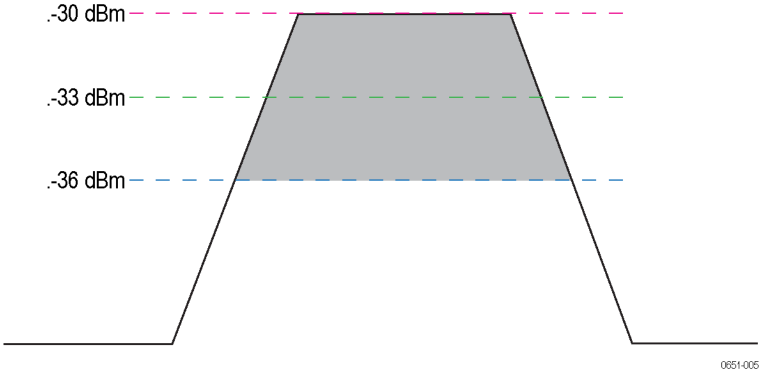
dBpk (dB below peak) 3.000
□ Automatic (50% or 3dB below peak)

NOTE. *Pulse criteria is assumed to be specifed as "dB below peak." Normally the specified value will be positive. For instance, if the peak value is -30dBm and the criteria is 3dB, then the pulse criteria will be -33dB during the measurement. Likewise, if 6dB is chosen, then the pulse criteria will be -36dB. As long as the overshoot is minimal and rise time is relatively steep, the automatic criteria shown above is adequate for most applications*

The figures below are intended to help clarify pulse criteria.



In figure 2, the peak value is -30dBm and the pulse criteria is 3dB. The shaded area represents the portion of the pulse that will be used to determine pulse power and duty cycle.



In Figure 3, the peak value remains -30dBm. The darkly shaded area represents the portion of the pulse that will be used to determine pulse power and duty cycle using a 6 dB peak criteria. Note that the average in Figure 3 will be lower than the average in Figure 2.

Pass Parameters:

Address, value of the peak criteria in dB below peak

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Pulse Setup

(See page 12, *Pulse Setup Command Group*.)

Sample Code Declarations:

C++	LB_API2 long _stdcall LB_SetPulseCriteria(long addr, double val); LB_API2 long _stdcall LB_GetPulseCriteria(long addr, double* val);
VB 6.0	Public Declare Function LB_SetPulseCriteria _ Lib "LB_API2.dll" (_ ByVal addr As Long, _ ByVal val As Double) _ As Long Public Declare Function LB_GetPulseCriteria _ Lib "LB_API2.dll" (_ ByVal addr As Long, _ ByRef val As Double) _ As Long

VB.NET

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

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

C#

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

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

LB_SetPulseReference (and related commands)

Related Commands:

LB_GetPulseReference

These commands configure the instrument for relative measurements during pulse measurements. (Other commands set a reference for CW measurements.) When making relative measurements, the units of measure must be set to "dB Relative".

For more information, see the commands LB_SetMeasurementPowerUnits and LB_GetMeasurementPowerUnits. The reference may be changed during a relative measurement. All relative measurements are made as a ratio and reported as dB above or below the reference.

Other commands of interest:

LB_SetMeasurementPowerUnits

LB_GetMeasurementPowerUnits

Pass Parameters:

Address, reference level, power units. The units enumeration is shown below:

DBM = 0 ' dBm

DBW = 1 ' dBW

DBKW = 2 ' dBkW

DBUV = 3 ' dBuV

W = 4 ' Watts

V = 5 ' Volts

DBREL = 6 ' dB Relative ***INVALID FOR SETTING A REFERENCE***

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Pulse Setup

(See page 12, *Pulse Setup Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetPulseReference(long addr,
    double pulseRef,
    double peakRef,
    double averageRef,
    double dutyCycleRef,
    PWR_UNITS units);

LB_API2 long _stdcall LB_GetPulseReference(long addr,
    double* pulseRef,
    double* peakRef,
    double* averageRef,
    double* dutyCycleRef,
    PWR_UNITS* units);
```

VB 6.0

```
Public Declare Function LB_SetPulseReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal pulseRef As Double, _
    ByVal peakRef As Double, _
    ByVal averageRef As Double, _
    ByVal dutyCycleRef As Double, _
    ByVal units As PWR_UNITS) _
    As Long

Public Declare Function LB_GetPulseReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef pulseRef As Double, _
    ByRef peakRef As Double, _
    ByRef averageRef As Double, _
    ByRef dutyCycleRef As Double, _
    ByRef units As PWR_UNITS) _
    As Long
```

VB.NET

```
Public Declare Function LB_SetPulseReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal pulseRef As Double, _
    ByVal peakRef As Double, _
    ByVal averageRef As Double, _
    ByVal dutyCycleRef As Double, _
    ByVal units As PWR_UNITS) _
    As Integer

Public Declare Function LB_GetPulseReference _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef pulseRef As Double, _
    ByRef peakRef As Double, _
    ByRef averageRef As Double, _
    ByRef dutyCycleRef As Double, _
    ByRef units As PWR_UNITS) _
    As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetPulseReference(
    int addr,
    double pulseRef,
    double peakRef,
    double averageRef,
    double dutyCycleRef,
    PWR_UNITS units );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetPulseReference(
    int addr,
    ref double pulseRef,
    ref double peakRef,
    ref double averageRef,
    ref double dutyCycleRef,
    ref PWR_UNITS units );
```

LB_SetResponseEnabled (and related commands)

Related Commands:

LB_GetResponseEnabled

LB_SetResponse

LB_GetResponse

These commands set the response, and enable/disable the feature.

Response is a frequency sensitive offset; as the measurement frequency is changed, the response changes accordingly. Response amplitude is always expressed in dB and the frequency is expressed in Hz. The interpolation is linear with respect to frequency and dB.

The response allows up to 201 points to be entered. The response points are frequency and amplitude pairs. Each set of function calls below are accompanied by the definition of the points. When the points are passed, the number of points must be specified as well.

Setting the response is independent of enabling or disabling the feature.

Pass Parameters:

Address, array of response structures, number of response structures (1 to 201)

OR

Address, feature state (1 = on, 0 = off)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Setup

(See page 19, *Setup Command Group*.)

Sample Code Declarations:

C++

```
struct ResponsePoints
{
    double frequency;
    double amplitude;
};

LB_API2 long _stdcall LB_SetResponseEnabled(
    long addr,
    FEATURE_STATE st);

LB_API2 long _stdcall LB_SetResponse(
    long addr,
    ResponsePoints* pts,
    long NumPts);

LB_API2 long _stdcall LB_GetResponseEnabled(
    long addr,
    FEATURE_STATE* st);

LB_API2 long _stdcall LB_GetResponse(
    long addr,
    ResponsePoints* pts,
    long* NumPts);
```

VB 6.0

```
Public Type ResponsePoints
    Frequency As Double
    Amplitude As Double
End Type

Public Declare Function LB_SetResponse _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef pts As ResponsePoints, _
    ByVal numPts As Long) _
    As Long

Public Declare Function LB_GetResponse _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef pts As ResponsePoints, _
    ByRef numPts As Long) _
    As Long

Public Declare Function LB_SetResponseEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal st As FEATURE_STATE) _
    As Long

Public Declare Function LB_GetResponseEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef st As FEATURE_STATE) _
    As Long
```

VB.NET

```
Public Structure ResponsePoints
    Dim Frequency As Double
    Dim Amplitude As Double
End Structure

Public Declare Function LB_SetResponse _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef pts As ResponsePoints, _
    ByVal numPts As Integer) _
    As Integer

Public Declare Function LB_GetResponse _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef pts As ResponsePoints, _
    ByRef numPts As Integer) _
    As Integer
```

```
Public Declare Function LB_SetResponseEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal st As FEATURE_STATE) _
      As Integer

Public Declare Function LB_GetResponseEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByRef st As FEATURE_STATE) _
      As Integer
```

C#

```
public struct ResponsePoints
{
      public double Frequency;
      public double Amplitude;
}

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetResponse(
      int addr,
      ref ResponsePoints pts,
      int numPts );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetResponse(
      int addr,
      ref ResponsePoints pts,
      ref int numPts );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetResponseEnabled(
      int addr,
      FEATURE_STATE st );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetResponseEnabled(
      int addr,
      ref FEATURE_STATE st );
```

LB_SetTTLTriggerInEnabled (and related commands)

Related Commands:

LB_GetTTLTriggerInEnabled

LB_SetTTLTriggerInInverted

LB_GetTTLTriggerInInverted

LB_SetTTLTriggerInTimeOut

LB_GetTTLTriggerInTimeOut

These commands control or read back the state of the external trigger input. The trigger-in features are available on instruments with the option present. The trigger is assumed to be TTL compatible, and positive edge triggered. The trigger-in can be enabled, disabled or inverted; or the timeout value can be set or read.

The trigger-in defines or controls the start of a measurement cycle. After the trigger is detected, the measurement will commence and will continue for the specified number of averages. Once a measurement is requested (LB_MeasureCW, LB_MeasurePulse, etc.) the system will monitor the trigger-in port.

If a trigger is not detected in the allotted time, the system will time out and return an invalid measurement. The time out of the trigger is set using SetTTLTriggerInTimeOut and specified in milliseconds.

As stated previously, the trigger may be inverted. When the trigger-in is inverted, the system will look for a negative edge (instead of a positive edge) and begin the measurement when a negative edge is detected.

Other commands of interest:

LB_MeasureCW

LB_MeasureCW_PF

LB_MeasurePulse

LB_MeasurePulse_PF

Pass Parameters:

None

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Trigger

(See page 21, *Trigger Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_SetTTLTriggerInEnabled(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_SetTTLTriggerInInverted(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_SetTTLTriggerInTimeOut(
      long addr,
      long val);

LB_API2 long _stdcall LB_GetTTLTriggerInEnabled(
      long addr,
      FEATURE_STATE* st);

LB_API2 long _stdcall LB_GetTTLTriggerInTimeOut(
      long addr,
      long val);

LB_API2 long _stdcall LB_GetTTLTriggerInInverted(
      long addr,
      FEATURE_STATE* st);
```

VB 6.0

```
Public Declare Function LB_SetTTLTriggerInEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal st As FEATURE_STATE) _
      As Long
Public Declare Function LB_SetTTLTriggerInInverted _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal st As FEATURE_STATE) _
      As Long
Public Declare Function LB_SetTTLTriggerInTimeOut _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal timeOut As Long) _
      As Long
Public Declare Function LB_GetTTLTriggerInEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByRef st As FEATURE_STATE) _
      As Long
```

```
Public Declare Function LB_GetTTLTriggerInInverted _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByRef st As FEATURE_STATE) _
       As Long
Public Declare Function LB_GetTTLTriggerInTimeOut _
       Lib "LB_API2.dll" ( _
       ByVal addr As Long, _
       ByRef timeOut As Long) _
       As Long
```

VB.NET

```
Public Declare Function LB_SetTTLTriggerInEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByVal st As FEATURE_STATE) _
       As Integer
Public Declare Function LB_SetTTLTriggerInInverted _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByVal st As FEATURE_STATE) _
       As Integer
Public Declare Function LB_SetTTLTriggerInTimeOut _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByVal timeOut As Integer) _
       As Integer
Public Declare Function LB_GetTTLTriggerInEnabled _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef st As FEATURE_STATE) _
       As Integer
Public Declare Function LB_GetTTLTriggerInInverted _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef st As FEATURE_STATE) _
       As Integer
Public Declare Function LB_GetTTLTriggerInTimeOut _
       Lib "LB_API2.dll" ( _
       ByVal addr As Integer, _
       ByRef timeOut As Integer) _
       As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetTTLTriggerInEnabled(
       int addr,
       FEATURE_STATE st );
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetTTLTriggerInInverted(
       int addr,
       FEATURE_STATE st );
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetTTLTriggerInTimeOut(
       int addr,
       int timeOut );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetTTLTriggerInEnabled(
       int addr,
       ref FEATURE_STATE st );
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetTTLTriggerInInverted(
       int addr,
       ref FEATURE_STATE st );
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetTTLTriggerInTimeOut(
       int addr,
       ref int timeOut );
```

LB_SetTTLTriggerOutEnabled (and related commands)

Related Commands:

LB_GetTTLTriggerOutEnabled

LB_SetTTLTriggerOutInverted

LB_GetTTLTriggerOutInverted

These features apply only to those instruments that have the trigger option.

These commands control the trigger output of the device. The trigger-out is compatible with TTL levels. It can be enabled, disabled, inverted or normal. The trigger-out occurs at the beginning of a measurement.

This means that if the measurement is untriggered (i.e. trigger-in is disabled) and trigger-out is enabled, a trigger will be produced each time a measurement is made. If the trigger-in is enabled, the trigger will be passed through when it is received.

A trigger output is normally low. When a trigger is produced, it begins with a positive-going edge and stays at a TTL level for a few microseconds, then returns to ground potential. If the trigger-out is inverted, it will transition from a high to a low TTL level. When a trigger is produced, a negative edge will be produced going to a TTL low. After a few microseconds, it will return to a TTL high.

Pass Parameters:

Address, feature state (1 = on, 0 = off)

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Trigger

(See page 21, *Trigger Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_GetTTLTriggerOutEnabled(
      long addr,
      FEATURE_STATE* st);

LB_API2 long _stdcall LB_SetTTLTriggerOutEnabled(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_SetTTLTriggerOutInverted(
      long addr,
      FEATURE_STATE st);

LB_API2 long _stdcall LB_GetTTLTriggerOutInverted(
      long addr,
      FEATURE_STATE* st);
```

VB 6.0

```
Public Declare Function LB_SetTTLTriggerOutEnabled _
    Lib "LB_API2.dll" ( _ ByVal addr As Long, _
    ByVal st As FEATURE_STATE) _
    As Long

Public Declare Function LB_SetTTLTriggerOutInverted _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByVal st As FEATURE_STATE) _
    As Long

Public Declare Function LB_GetTTLTriggerOutEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef st As FEATURE_STATE) _
    As Long

Public Declare Function LB_GetTTLTriggerOutInverted _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long, _
    ByRef st As FEATURE_STATE) _
    As Long
```

VB.NET

```
Public Declare Function LB_SetTTLTriggerOutEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal st As FEATURE_STATE) _
    As Integer

Public Declare Function LB_SetTTLTriggerOutInverted _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByVal st As FEATURE_STATE) _
    As Integer

Public Declare Function LB_GetTTLTriggerOutEnabled _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef st As FEATURE_STATE) _
    As Integer

Public Declare Function LB_GetTTLTriggerOutInverted _
    Lib "LB_API2.dll" ( _
    ByVal addr As Integer, _
    ByRef st As FEATURE_STATE) _
    As Integer
```

C#

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

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetTTLTriggerOutInverted(
    int addr,
    FEATURE_STATE st );

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

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetTTLTriggerOutInverted(
int addr,
ref FEATURE_STATE st );
```

LB_StoreReg (and related commands)

Related Commands:

LB_RecallReg

These commands are the traditional store/recall register commands. There are 20 registers and each register holds an entire state. Unlike most instruments, however, the states are NOT held in the instrument, but are stored on the local PC in an *.INI file (basic text file). The files are named by model number and address, and are retained in the Tektronix application directory.

This means that saved states can be saved, copied or moved between PCs. Any instrument that is initialized with that address will use the states with a properly named *.INI file. The naming convention of the file is as follows, where xxx is the address of the unit:

PSM3110_xxx.ini

PSM3120_xxx.ini

PSM3310_xxx.ini

PSM3320_xxx.ini

PSM3510_xxx.ini

PSM4110_xxx.ini

PSM4120_xxx.ini

PSM4410_xxx.ini

PSM4320_xxx.ini

PSM5110_xxx.ini

PSM5120_xxx.ini

PSM5410_xxx.ini

PSM5320_xxx.ini

More importantly, the names of the files may be renamed and managed by the user application (perhaps stored in a data base as a long string); the store/recall registers are now under user control.

Pass Parameters:

None

Returned Values:

Success: > 0

Error: <= 0

Command Group:

Save/Recall

(See page 18, *Save/Recall Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_Store(
      long addr,
      long regIdx);

LB_API2 long _stdcall LB_Recall(
      long addr,
      long regIdx);
```

VB 6.0

```
Public Declare Function LB_Store _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal regIdx As Long) _
      As Long

Public Declare Function LB_Recall _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal regIdx As Long) _
      As Long
```

VB.NET

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

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

C#

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

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

LB_WillAddressConflict

This command checks the address of all instruments connected to the system. If any of the addresses match, a conflict is deemed to exist. If all the addresses are unique to the system, a conflict is deemed not to exist.

Pass Parameters:

None

Returned Values:

Conflict Exists = 1

Conflict does not exit = 0

Error < 0

Command Group:

Initialization and Identification

(See page 10, *Initialization and Identification Command Group*.)

Sample Code Declarations:

C++

```
LB_API2 long _stdcall LB_WillAddressConflict(long addr);
```

VB 6.0

```
Public Declare Function LB_WillAddressConflict _
    Lib "LB_API2.dll" ( _
    ByVal addr As Long) _
    As Long
```

VB.NET

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

C#

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

PP_AnalysisTraceIsValid

This command 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 commands operate on the analysis trace. An analysis trace is most commonly obtained by calling PP_CurrTrace2AnalysisTrace after having taken a trace (see PP_GetTrace).

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Status

(See page 16, *Pulse Profiling Status Command Group*.)

Sample Code 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#

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

PP_CheckTrigger

This command 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 instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Status

(See page 16, *Pulse Profiling Status Command Group*.)

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

PP_CnvtTrace

This command 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
}
```

Pass Parameters:

addr – address of the selected instrument

*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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code 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#

```
[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

This command copies the current trace to the analysis trace and returns a copy of that trace.

The driver potentially holds 2 traces for each initialized instrument. One trace is the current trace; the second is the analysis trace. The current trace is the most recently taken trace. The analysis trace is the trace data used to make measurements.

Pass Parameters:

addr – address of the selected instrument

*tr – pointer to an array of doubles

trLen – the length of the trace

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

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

PP_GatePositionIsValid

This command determines whether 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 instrument

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.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetAnalysisTraceLength

This command returns a 32 bit integer indicating the length of the analysis trace. The minimum timeout is 10,000 µs or 10 ms. The maximum timeout is 11 seconds or 11,000,000 µs.

Pass Parameters:

- addr is a 32 bit integer containing the address of the device for which the length of the analysis trace is desired

Returned Values:

A return value of greater than zero indicates success and the length of the analysis trace. The analysis trace can vary from 480 to 10000 points depending on sweep time. A return value less than 0 indicates failure.

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code Declarations:

C++	Long PP_GetAnalysisTraceLength(long addr);
VB.NET	Public Declare Function GetAnalysisTraceLength Lib "LB_API2.dll" (ByVal addr As Integer) As Integer
C#	[System.Runtime.InteropServices.DllImport("LB_API2.dll")] public static extern int PP_GetAnalysisTraceLength(int addr);

PP_GetGateCrestFactor

This command returns the create factor (in dB) of the span in the analysis trace specified by the gate.

Pass Parameters:

addr – address of the selected instrument

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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGateDroop

This command 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 instrument

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.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGateDutyCycle

This command returns the duty cycle (as a decimal) of span in the analysis trace specified by the gate.

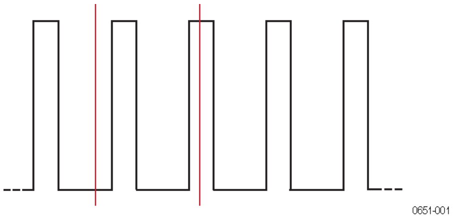
Pass Parameters:

addr – address of the selected instrument

gateIdx – index of the selected gate. The 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.



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGateEndPosition

This command 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 instrument

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).

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGateFallTime

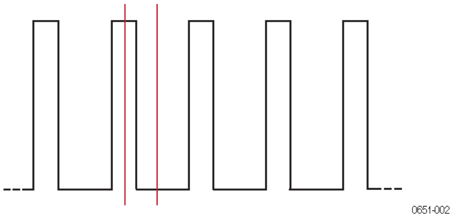
This command returns the fall time in microseconds of the pulse delineated by the selected gate. The gate must be properly positioned to return a proper value. The left side of the gate must be positioned between a pulse rising and falling edge. The right side must be positioned after the next falling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

Pass Parameters:

addr – address of the selected instrument

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.



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGateOverShoot

This command 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 instrument

gateIdx – index of the selected gate

*overShoot –overshoot in dB as outlined above

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGatePeakPower

This command returns the peak power measured of the analysis trace as defined by the gate edges.

Pass Parameters:

addr – address of the selected instrument

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.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGatePRF

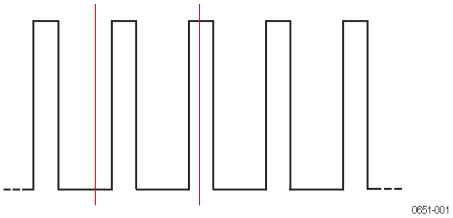
This command returns the pulse repetition frequency (PRF) in Hertz, as defined by the inverse of the time between the rising edges of the first two complete pulses present in the span defined by the gate (gateIdx). A complete pulse is a rising edge followed by falling edge. If two complete pulses are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

Pass Parameters:

addr – address of the selected instrument

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.



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGatePRT

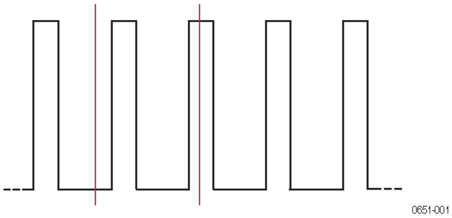
This command returns the pulse repetition time (PRT) in microseconds using the same algorithm defined for PRF. The sole difference is that time instead of frequency is returned.

Pass Parameters:

addr – address of the selected instrument

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.



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code Declarations:

C++

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

VB.NET

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

C#

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

PP_GetGatePulsePower

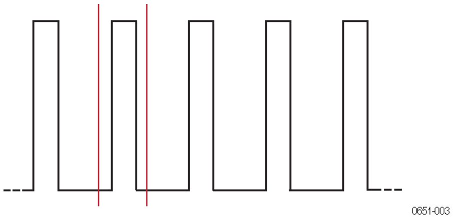
This command returns average pulse power.

Pass Parameters:

addr – address of the selected instrument

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



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

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

PP_GetGatePulseWidth

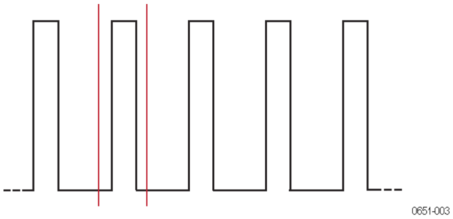
This command measures the pulse width in microseconds.

Pass Parameters:

addr – address of the selected instrument

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



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code 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#

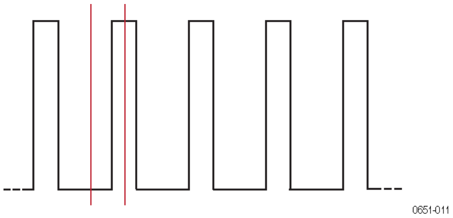
```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetGatePulseWidth
    (int addr,
    int gateIdx,
    ref double plsWidth);
```

PP_GetGateRiseTime

This command returns rise time in microseconds.

Pass Parameters:

addr – address of the selected instrument *riseTm – Measured rise time in microseconds. The gate edges must be set as shown below.



Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

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

PP_GetMarkerAmp

This command returns the amplitude of the trace at the point indicated by the marker.

Pass Parameters:

addr – address of the selected instrument

mrkIdx – index of marker (0..4)

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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

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

PP_GetMarkerDeltaAmp

This command returns the difference in amplitude between the normal marker and the delta marker in dBm.

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

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

PP_GetPeaks_Val (and related commands)

Related Commands:

PP_GetPeaks_Idx

PP_GetPeaksFromTr_Val

PP_GetPeaksFromTr_Idx

PP_GetPeaks_VEE_Idx

PP_GetPeaks_VEE_Val

These commands return a set of peaks from either the analysis trace (PP_GetPeaks_Val and PP_GetPeaks_Idx) or from a trace passed to the command. The more complex commands have the added advantage that the trace may be any compatible trace, and can use a peak criteria and threshold different from 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 the 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 a minimum of two points or pixels is required for each peak.

The _VEE calls are designed to be used in the VEE programming environment which does not allow for arrays of structures. Instead of an array of Peak structures, the _VEE calls pass an array of longs and doubles.

Pass Parameters:

addr – address of the selected instrument

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 command.

peakCrit – peak criteria used to define a peak.

measThresh – measurement threshold use d to filter peaks.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code 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#

```
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_GetPulseEdgesTime (and related commands)

Related Commands:

PP_GetPulseEdgesPosition

These commands return the index of the leading and trailing edges of the pulse containing the peak defined by pkTime or pkIdx. These calls are intended to be used with PP_GetPeak and other commands 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 is that everything is in time (microseconds) instead of trace index.

1. Acquire a trace (PP_GetTrace)
2. Move current trace to analysis trace (PP_CurrTrace2AnalysisTrace)
3. Get the peaks from the trace sorted by index (PP_GetPeaks_Idx)
4. 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.
5. Select the peaks of interest (pick the first peak returned in P_GetPeaks_Idx)
6. Get the edges of the pulses containing the peak (PP_GetPulseEdgesPosition)
7. Set the mode of the selected gate to ON (PP_SetGateMode)
8. Set the edges of the gate appropriately for the measurement: (PP_SetGateStartEndPosition)
   - For rise time, set the left gate edge before the rising edge. Set the right gate edge midway between the rising and falling edges.

     Example:
     Assume a 1ms sweep time (10,000 points) for a resolution of 100 ns.
     Assume a 10 kHz signal with a 20% duty cycle.
     Assume the 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 left pulse edge will be about 1000, and the right pulse edge will be about 1200.
   - Set the gate edges so that the left side of the gate is at 950 (about 50 pixesl before the rising edge), and the right side of the gate is at 1100 (midway between rising and falling edge).
9. You can now measure the rise time using PP_GetGateRiseTime.

NOTE. *This function and the Related Calls: are especially useful in making programmatic measurements. These functions allow for the easiest placement of gate edges.*

Pass Parameters:

addr – address of the selected instrument

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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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_GetTrace

This command causes the instrument 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 instrument, 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:

1. Initialize the instrument (LB_Initialize_Addr)
2. Set the frequency (LB_SetFrequency)
3. Set the sweep time (PP_SetSweepTime)
4. Get the length of the trace (PP_GetTraceLength)
5. Allocate an array equal to or larger than trace length
6. Get a trace (PP_GetTrace)
7. Move the current trace to the analysis trace (PP_CurrTrace2AnalysisTrace)
8. Get the peaks orders by index (PP_GetPeaks_Idx)
9. Use the first two peaks from the previous call to get pulse edges (PP_GetPulseEdgesPosition)
10. Set the mode of the gate to ON(PP_SetGateMode)
11. 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).
12. Make the PRF measurement (PP_GetGatePRF)

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

1. Get a trace (PP_GetTrace)
2. Move the current trace to the analysis trace (PP_CurrTrace2AnalysisTrace)
3. Make the PRF measurement (PP_GetGatePRF)

This short sequence makes several assumptions; first, 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:

1. Initialize the instrument (LB_Initialize_Addr)
2. Set the frequency (LB_SetFrequency)
3. Set the sweep time (PP_SetSweepTime)
4. Get the length of the trace (PP_GetTraceLength)
5. Allocate an array equal to or larger than trace length
6. Get a trace (PP_GetTrace)
7. Move the current trace to the analysis trace (PP_CurrTrace2AnalysisTrace)
8. Get the peaks orders by index (PP_GetPeaks_Idx)
9. Use the first two peaks from the previous command to get pulse edges (PP_GetPulseEdgesPosition)

10. Set the mode of the gate to ON (PP_SetGateMode)
11. 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).
12. Make the PRF measurement (PP_GetGatePRF)
13. Using the current gate and trace make a PRT measurement (PP_GetGatePRT)
14. Using the current gate and trace make a pulse width measurement (PP_GetGatePulseWidth)
15. Using the same pulse edge information reposition the gate edges for a rise time measurement (SetGateStartEndPosition).
16. Make a rise time measurement (PP_GetGateRiseTime)

Since steps x through y may be used often, it may be useful to combine them as a function.

[Step x is "Set the frequency". Step y is "Use the first two peaks from the previous...]

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 instrument

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

Returned Values:

Failure < 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code 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#

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

PP_GetTraceAvgPower (and related commands)

Related Commands:

PP_GetTraceCrestFactor

PP_GetTraceDC

PP_GetTracePkPwr

PP_GetTracePulsePower

These commands make a number of measurements similar to the power meter measurement commands, but operate on a single trace (which may or may not be averaged) instead of a set of random samples. These measurement results may differ from the gated measurements. Gate measurements require the user to select a particular cycle within a trace.

Normally, these differences are unimportant. However, there are times when these distinctions account for differences in the measurements. It should be noted that these differences are not errors; rather, the variations in results 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 (give a specific set of samples), random samples may not always represent the best means of collecting the data for the user's 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 the user to select and measure a specific portion of the signal, and at the same time ignore all other data. It is critical 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 cases where additional assurance is desirable, use the power meter measurements along side gates measurements, or use these-trace base measurements.

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code 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#

```
[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_GetTraceLength

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

Sweep Time	# Trace Points	Over Sampling	Resolution (time/points)
10 µs	480	96	0.02833 µs
20 µs	960	96	0.02833 µs
50µs	2400	96	0.02833 µs
100 µs	4800	96	0.02833 µs
200 µs	9600	96	0.02833 µs
500 µs	10,000	48	0.05000 µs
1,000 µs	10,000	24	0.10000 µs
2,000 µs	10,000	24	0.20000 µs
5,000 µs	10,000	24	0.50000 µs
10,000 µs	10,000	24	1.00000 µs
20,000 µs	10,000	12	2.00000 µs
50,000 µs	10,000	6	5.00000 µs
100,000 µs	10,000	2	10.00000 µs
200,000 µs	10,000	1	20.00000 µs
500,000µs	10,000	1	50.00000 µs
1,000,000 µs	10,000	1	100.00000 µs

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code 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#

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

PP_MarkerPosIsValid

This command 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, 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 instrument

mrkIdx – index of marker (0..4)

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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

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

PP_MarkerToPk (and related commands)

Related Commands:

PP_MarkerToLowestPk

PP_MarkerToFirstPk

PP_MarkerToLastPk

PP_MarkerPrevPk

PP_MarkerNextPk

PP_MarkerPkHigher

PP_MarkerPkLower

These commands set one of five markers (0<=mrkIdx<=4) to the position specified in the command. The underlying algorithm begins by getting a list of the peaks ordered by index or value. The subsequent actions 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. *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 instrument

mrkIdx – index of the marker (0..4)

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

```
[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_SetAnalysisTrace (and related commands)

Related commands:

PP_GetAnalysisTrace

These commands are companions to PP_CurrTrace2AnalysisTrace. While CurrTrace2AnalysisTrace copies the current trace to the analysis trace, these commands allow you to get and set the analysis trace directly. Bear in mind, when using these functions you must also know the frequency, sweep time, trace length and units of measure (should be dBm).

Pass Parameters:

addr – address of the device

frequency – in Hertz

sweepTime – in microseconds

tr – an array or pointer to an array of 64-bit double precision floating points

trLen – length of the trace

PWR_UNITS – should be set to DBM (or 0)

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trace

(See page 16, *Pulse Profiling Trace Command Group*.)

Sample Code Declarations:

C++

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

```
long PP_GetAnalysisTrace(long addr, double* frequency, double* sweepTime, double*tr, long* trLen,
PWR_UNITS* units)
```

VB.NET

```
Public Declare Function PP_SetAnalysisTrace Lib "LB_API2.dll"
         (int addr,
        double frequency,
        double sweepTime,
        double tr(),
        int trLen,
        PWR_UNITS units)
       As Integer

Public Declare Function PP_GetAnalysisTrace Lib "LB_API2.dll"
        (int addr,
        double frequency,
        double sweepTime,
        double tr(),
        int trLen,
        PWR_UNITS units)
        As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int PP_SetAnalysisTrace(int addr,
        double frequency,
        double sweepTime,
        double[] tr,
        int trLen,
       PWR_UNITS units)

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
int PP_GetAnalysisTrace(int addr,
        double frequency,
        double sweepTime,
        double[] tr,
        int trLen,
        PWR_UNITS units)
```

PP_SetAvgMode (and related commands)

Related Commands:

PP_GetTraceAvgs

PP_GetAvgMode

PP_ResetTraceAveraging

These commands set, auto-set or manual reset the averaging mode.

Trace averaging can be very important to making good measurements, and will reduce the noise on the trace. There are three elements to trace averaging:

- setting the mode
- selecting the number of traces to average
- controlling the current state of averaging

PP_SetAvgMode sets the current trace averaging mode. The averaging mode may be off, auto-reset or manual reset. If the 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 the 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_GetAvgMode gets the current trace averaging mode.

PP_GetTraceAvgs 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 instrument

*mode – pointer to AVG_MODE (32 bit integer)

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
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 commands)

Related Commands:

PP_GetAvgResetSens

These commands set or get the criteria used to reset the averaging when the averaging mode is AVG_AUTO_RESET (see PP_SetAvgMode and PP_GetAvgMode).

Pass Parameters:

addr – address of the selected instrument

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

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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_SetClosestSweepTimeUSEC

This command sets the sweep time to the fixed sweep time closest to the sweep time sent (in microseconds) to the command. For instance, if a value of 11 was sent (meaning 11 µs sweep time) the system would set the sweep time to 10 µs .

Pass Parameters:

addr – address of the selected instrument

swpTm – desired sweep time in microseconds

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

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

PP_SetFilter (and related commands)

Related Commands:

PP_GetFilter

These commands set or get the enumeration associated with the current filter settings. The enumeration 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 instrument
fltrIdx – index of cutoff frequency
fltrPolse – index of filter poles

Returned Values:

Failure <= 0
Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
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_SetGateMode (and related commands)

Related Commands:

PP_GetGateMode

These commands set or get 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 instrument

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.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
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_SetGateStartEndPosition (and related commands)

Related Commands:

PP_GetGateStartEndPosition

PP_SetGateStartEndTime

PP_GetGateStartEndTime

PP_SetGateStartPosition

PP_GetGateStartPosition

PP_SetGateEndPosition

PP_GetGateEndPosition

PP_SetGateStartTime

PP_GetGateStartTime

PP_SetGateEndTime

PP_GetGateEndTime

These commands set or get the gate start (left side) and/or end (right side) 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. Time is in microseconds. Index is trace index.

Pass Parameters:

addr – address of the selected instrument

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)

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Gate

(See page 12, *Pulse Profiling Gate Command Group*.)

Sample Code Declarations:

C++

```
long __stdcall PP_SetGateStartEndPosition(long addr,
    long gateIdx,
    long sttIdx,
    ong 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#

```
[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_SetMarkerDeltaTime (and related commands)

Related Commands:

PP_GetMarkerDeltaTime

These commands set or get the positions the selected marker in microseconds.

Pass Parameters:

addr – address of the selected instrument

mrkIdx – index of marker (0..4)

mrkTm – time in microseconds

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

```
[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 commands)

Related Commands:

PP_GetMarkerMode

These commands set or get the marker mode to on, normal or delta marker.

Pass Parameters:

addr – address of the selected instrument

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.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

```
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 commands)

Related Commands:

PP_GetMarkerPosition

PP_SetMarkerPositionTime

PP_GetMarkerPositionTime

These commands set or get 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 instrument

mrkIdx – index of marker

trIdx or mrkTm –trace index or time in microseconds

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Marker

(See page 14, *Pulse Profiling Marker Command Group*.)

Sample Code 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#

```
[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,
    ef 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 commands)

Related Commands:

PP_GetMeasurementThreshold

These commands set or get the measurement threshold. The measurement threshold, along with the peak criteria, affects a number of measurement commands, especially the peak commands. 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 instrument

measThreshold_dBm – measurement threshold in dBm

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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_SetPoles (and related commands)

Related Commands:

PP_GetPoles

These commands set or get the number of poles in the current filter. As the number of poles increase, the sharpness of the cutoff increases. The valid indices are 0…2 indicating the number of poles between 1..4.

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0 Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
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_SetSweepDelay (and related commands)

Related Commands:

PP_GetSweepDelay

These commands set or get 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 Sweep Time (Under sampled)	Max Sweep time (No under sampling)
10 µs to 10 ms	1 <= 10 ms	
20 ms to 50 ms	1 <= 10 ms	>10 ms to 999 ms
100 ms to 1 second		>10 ms to 999 ms

Delay sweep is taken in one of two ways. Sweep times at 10 ms 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 instrument

SwpDly –delay in microseconds before data is taken. Delay is measured from the trigger edge.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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 (and related commands)

Related Commands:

PP_GetSweetDelayMode

These commands turn the sweep delay on or off. The sweep delay parameter remains unchanged.

Pass Parameters:

addr – address of the selected instrument

SwpDlyMode – 0=OFF, 1 = ON

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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 commands)

Related Commands:

PP_GetSweepHoldOff

These commands specify (or get) the length of time (in microseconds) to wait after a sweep or trace is taken.

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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 commands)

Related Commands:

PP_GetSweepTime

These commands set or get the sweep time (in microseconds) for the next sweep taken. Sweep time is a 1, 2, 5 sequence starting with 10 µs 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 µs	480	96	0.02833 µs
20 µs	960	96	0.02833 µs
50 µs	2400	96	0.02833 µs
100 µs	4800	96	0.02833 µs
200 µs	9600	96	0.02833 µs
500 µs	10,000	48	0.05000 µs
1,000 µs	10,000	24	0.10000 µs
2,000 µs	10,000	24	0.20000 µs
5,000 µs	10,000	24	0.50000 µs
10,000 µs	10,000	24	1.00000 µs
20,000 µs	10,000	12	2.00000 µs
50,000 µs	10,000	6	5.00000 µs
100,000 µs	10,000	2	10.00000 µs
200,000 µs	10,000	1	20.00000 µs
500,000 µs	10,000	1	50.00000 µs
1,000,000 µs	10,000	1	100.00000 µs

Pass Parameters:

addr – address of the selected instrument

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code 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#

```
[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_SetTimeOut (and related commands)

Related Commands:

PP_GetTimeOut

These commands set or get the timeout used while taking a trace.

Pass Parameters:

- addr is a 32 bit integer containing the address of the device for which the length of the analysis trace is desired
- tmoUSEC is a 32 bit integer indicating the timeout in microseconds.

Returned Values:

A return value of greater than zero indicates success. A return value less than zero indicates failure.

Command Group:

Pulse Profiling Setup

(See page 15, *Pulse Profiling Setup Command Group*.)

Sample Code Declarations:

C++

```
long PP_GetTimeOut(long addr, long* tmoUSEC);
long PP_SetTimeOut(long addr, long tmoUSEC);
```

VB.NET

```
Public Declare Function int PP_GetTimeOut Lib "LB_API2.dll" (ByVal addr As Integer, ByRef tmoUSEC
as integer) As Integer
Public Declare Function int PP_SetTimeOut Lib "LB_API2.dll" (ByVal addr As Integer, ByVal tmoUSEC
as integer) As Integer
```

C#

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_GetTimeOut(int addr, ref int tmoUSEC);
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int PP_SetTimeOut(int addr, int tmoUSEC);
```

PP_SetTriggerEdge (and related commands)

Related Commands:

PP_GetTriggerEdge

These commands set or get 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 instrument

TrgEdge – specifies the trigger edge. This value can be 0 (positive) or 1 (negative)

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trigger

(See page 18, *Pulse Profiling Trigger Command Group*.)

Sample Code 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#

```
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 commands)

Related Commands:

PP_GetTriggerLevel

These commands set or get 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 instrument

TrgLvl – the trigger level value in dBm.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trigger

(See page 18, *Pulse Profiling Trigger Command Group*.)

Sample Code 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#

```
[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 commands)

Related Commands:

PP_GetTriggerOut

These commands set or get 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 instrument

trgOutMode – sets or gets the mode.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trigger

(See page 18, *Pulse Profiling Trigger Command Group*.)

Sample Code 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_
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#

```
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_SetTriggerSource (and related commands)

Related Commands:

PP_GetTriggerSource

These commands set or get the trigger source. Trigger source can be internal or external. External triggers are received via the SMB connector on the back of the instrument. External triggers are TTL triggers. They must have the following characteristics:

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

Internal triggers are derived from the incoming signal (most like an oscilloscope's 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 instrument TrgSrc – the trigger source, internal auto-level, internal manual level and external.

Returned Values:

Failure <= 0

Success >= 1

Command Group:

Pulse Profiling Trigger

(See page 18, *Pulse Profiling Trigger Command Group*.)

Sample Code 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#

```
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);
```

Index