UNIVERSAL SERIAL BUS POWER SENSOR

MODEL LB478A/LB479A/LB480A

LadyBug Technologies LLC

NOTICES

SAFETY

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

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

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

CONFORMITY

WEEE Compliant
RoHS Compliant
USB 2.0 Compliant

DISCLAIMER

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

WARRANTY

See the warranty section of the Product Manual for details.

DOCUMENT NUMBER

Not Assigned (Reference LB4XXA Programming Guide).

CONTACT INFORMATION

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

TABLE OF CONTENTS

Introduction

This is a preliminary programming guide for the LB478A and LB479A. This guide is generally applicable to the LB480A. At this juncture the programmatic interface consists of a dynamic link library or DLL. The name of the DLL is LB_API2.DLL. This library uses the WinAPI or "_stdcall" calling convention. We have chosen a DLL and this calling convention because they provide greater access to more of the most common environments. This DLL is located in the Ladybug application directory. The name of the default application directory is "C:\Program Files\Ladybug\LB479A".

Included in the installation is a demonstration program. The program is written in VB 6.0 and VB.NET. Almost all functions are demonstrated in this DLL. The name of the application is TestHarness2. A like-named sub-directory is located in the application directory. A header file and lib file are also included (LB_API2.h and LB_API2.lib). The DLL is compiled using Visual Studio 2005 Visual C++.

Making a Simple Measurement

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

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

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

Writing the Code:

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

- cmdGetAddress
- cmdInitialize
- cmdMeasure

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

Explanation of the Code:

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

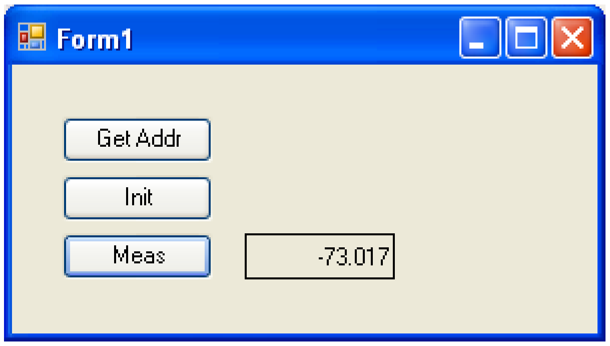
We can initialize the sensor using the address from the first call. This is accomplished by clicking the second button on the form. This makes the call "LB_InitializeSensor_Addr". This call can be interpreted as "initialize the sensor using the address". Initialization causes the calibration constants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

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

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

Using the Application:

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



Then follow the sequence outlined below:

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

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

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

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

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

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

VB 6.0 Code

```
Option Explicit

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

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

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

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

Dim m_Addr As Long

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

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

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

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

VB.NET Code (Visual Studio 2005)

```
Public Class Form1

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

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

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

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

    Dim m_Addr As Integer

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

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

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

        Dim CW As Double, rslt As Long

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

C SHARP Code (Visual Studio 2005)

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

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

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

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

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

        public int m_Addr;

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

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

            double CW = 0; long rslt = 0;

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

Addressing and Communicating with Sensors

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

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

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

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

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

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

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

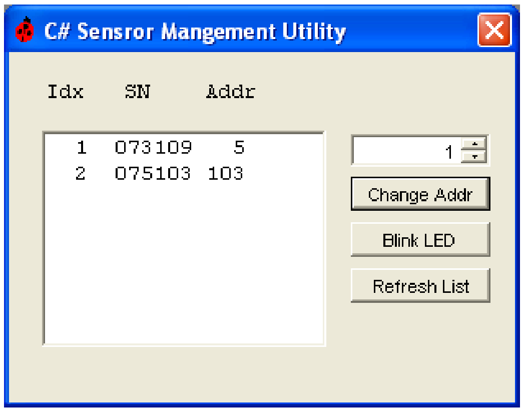
We will break this in to the following two steps:

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

Step 1 - Setting the Address(es)

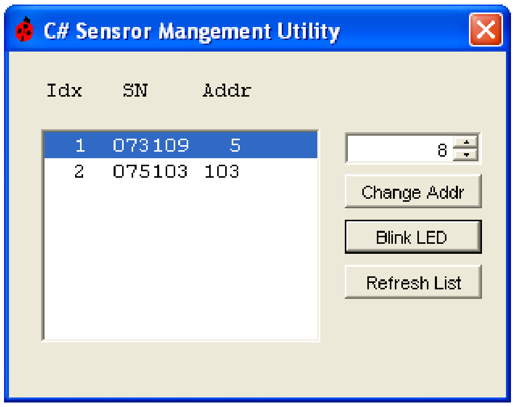
Open the “Managing Addresses” application provided to accomplish the first step. This application should be visible in the Ladybug menu (*Start* > *Ladybug* > *Addresses*). You should see the window below when the application starts.

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



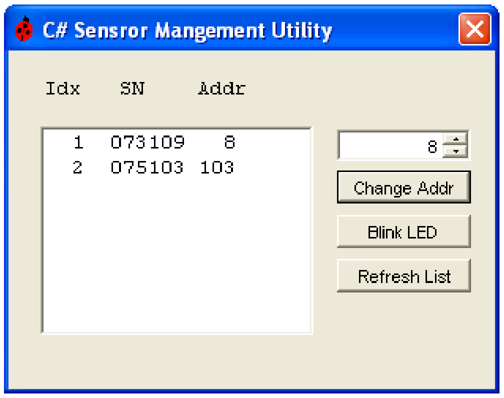
```
  1  073109    5
  2  075103  103
```

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



```
  1  073109    5
  2  075103  103
```

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



```
Idx   SN      Addr

1   073109    8
2   075103  103
```

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

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

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

Step 2 - Communicating with Your Sensor(s)

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

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

More Detail

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

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

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

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

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

The functions applicable to index are listed below:

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

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

The functions applicable to serial number are listed below:

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

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

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

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

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

User Function Calls

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

LIBRARY "LB_API2"

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

The driver is installed using a .inf file. The supplied LB4XX_2K.inf file is in the Ladybug Technologies LLC\LB479A sub-directory of the install directory.

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

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

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

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

SensorCnt

Description:

Returns the number of sensors currently connected to the computer.

Pass Parameters:

None

Return Value:

Success: The number of sensors connected to the PC. The number will be between 0 and 16
Failure: Any number < 0

Sample Code: (All examples)

Declarations:

C++

```
LB_API2 long _stdcall LB_SensorCnt();
```

VB 6.0

```
Public Declare Function LB_SensorCnt Lib "C:\LB_APPS\LB_API2\Debug\LB_API2.dll"
() As Long
```

VB.NET

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

C SHARP

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

SensorList

Description:

Returns a description for each sensor. The user must ensure that an array of sensor 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. There is code for VB.6 and VB.Net.

Pass Parameters:

A properly sized array of sensor descriptions.

Return Value:

Success: > 0, 1 plus the number if items
Failure: < 0

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 SHARP

```
      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 );
```

Get(Set)Address_SN

Description:

Returns the address given the serial number.

Pass Parameters:

Serial number, six characters in length plus one character for the zero termination is passed in both cases. In LB_SetAddress_SN the address is also passed.

Return Value:

Success: > The address between 1 and 255
Failure: < 0

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 SHARP

```
[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 );
```

Get(Set)Address_Idx

Description:

Returns the address given the index. Index is assigned by the OS when the unit is plugged in.

Pass Parameters:

Index 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.

Return Value:

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

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 SHARP

```
[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 );
```

ChangeAddress

Description:

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

Return Value:

Success: > 0
Failure: < 0

Declarations:

C++

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

VB 6.0

```
Public Declare Function LB_ChangeAddress _
      Lib "C:\LB_APPS\LB_API2\Debug\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 SHARP

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

AddressConflictExists

Description:

Checks the address of all sensors 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

Return Value:

Conflict Exists = 1
Conflict does not exit = 0
Error < 0

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 SHARP

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

WillAddressConflict

Description:

Checks the address of all sensors 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

Return Value:

Conflict Exists = 1
Conflict does not exit = 0
Error < 0

Declarations:

C++

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

VB 6.0

```
Public Declare Function LB_WillAddressConflict _
      Lib "C:\LB_APPS\LB_API2\Debug\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 SHARP

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

GetModelNumber_SN (_Idx)(_Addr)

Description:

These routines 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.

Return Value:

Returned values are -1 to 3 as denoted by the enumerations

Declarations:

C++

```
enum MODEL_NUMBER {
      LBUknwn = -1,
      LB4xxA  = 0,
      LB478A      = 1,
      LB479A      = 2,
      LB480A      = 3
};

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
    LBUknwn = -1
    LB4xxA = 0
    LB478A = 1
    LB479A = 2
    LB480A = 3
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
      LBUknwn = -1
      LB4xxA = 0
      LB478A = 1
      LB479A = 2
      LB480A = 3
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 SHARP

```
public enum MODEL_NUMBER
{ //  Enumeration of model numbers
      LBUknwn = -1,
      LB4xxA = 0,
      LB478A = 1,
      LB479A = 2,
      LB480A = 3,
}

[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);
```

IsSensorConnected_SN(_Addr)

Description:

Determines if the specified sensor 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.

Return Value:

Serial Number is connected: 1
Serial Number is NOT connected: 0
Error < 0

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 SHARP
```
[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 );
```

GetSerNo_Idx(_Addr)

Description:

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

Pass Parameters:

Index or address

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

BlinkLED_Idx(_SN)(_Addr)

Description:

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

Pass Parameters:

Index, serial number or address

Return Value:
Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

GetIndex_SN(_Addr)

Description:

These routines return the index given the serial number or address.

Pass Parameters:

Serial number or address

Return Value:

Success: > Index greater than 0
Error: <= 0

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 SHARP

```
[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 );
```

InitializeSensor_SN(_Idx)(_Addr)

Description:

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

Pass Parameters:

Index, serial number or address

Return Value:

Success: > 0
Error: <= 0

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 "C:\LB_APPS\LB_API2\Debug\LB_API2.dll" ( _
                            ByVal sn As String) _
                            As Long

Public Declare Function LB_InitializeSensor_Idx _
                            Lib "C:\LB_APPS\LB_API2\Debug\LB_API2.dll" ( _
                            ByVal idx As Long) _
                            As Long

Public Declare Function LB_InitializeSensor_Addr _
                            Lib "C:\LB_APPS\LB_API2\Debug\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 SHARP

```
[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 );
```

MeasureCW

Description:

This routine 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 msec per buffer. Each buffer contains about 120 averages so that a measurement for 100 buffers (averaging set to 100) would take 30 to 100 msec. 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 calls 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

Return Value:
Success: > 0
Error: <= 0

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

MeasureCW_PF

Description:

This routine makes CW measurements and evaluates that 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, you should set up limits. 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 calls of interest in addition to the calls 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)

Return Value:

Success: > 0
Error: <= 0

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                        ' 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

    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
                                          ' not limited to the following reasons:
                                          '      - limits are not enabled
                                          '      - limits are not specified
                                          '      - valid measurement not made
    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 SHARP

```
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 );
```

MeasurePulse

Description:

This routine 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. The measurements are made using the number of buffers (averages) and the units specified earlier. Much of the test that applies to CW measurement time also applies to pulse measurements.

NOTE: The duty cycle is measured, it is not calculated. However, the Ladybug sensor supports the old style pulse measurement where the duty cycle is specificed by the user.

Again, these are direct measurements. So 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 nsec, the best measurements require some care. For best results, you can 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. If your pulse measurement is 6dB higher than the peak power with the power turned off - and other limits are not breached – you are in good shape.

Finally, as the duty cycle decreases you will need to increase averaging; and as PRF increases you can decrease the number of averages. 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 calls of interest in addition to the calls 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.

Return Value:

Success: > 0
Error: <= 0

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_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 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 SHARP

```
[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 );
```

MeasurePulse_PF

Description:

This routine makes pulse measurements just as LBMeasurePulse does. This is coupled with a pass/fail judgement like the LB_MeasureCW_PF does. 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.

- 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.

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Get(Set)Frequency

Description:

These routines set 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.

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Get(Set)Averages

Description:

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

Pass Parameters:

Address, averages (1 to 30000)

Return Value:

Success: > 0
Error: <= 0

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 SHARP
```
[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 );
```

Set(Get)AutoPulseEnabled

Description:

These routines enable (value = 1) or disable (value = 0) 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_GetPulseCriteria

Pass Parameters:

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

Return Value:

Success: > 0
Error: <= 0

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 SHARP

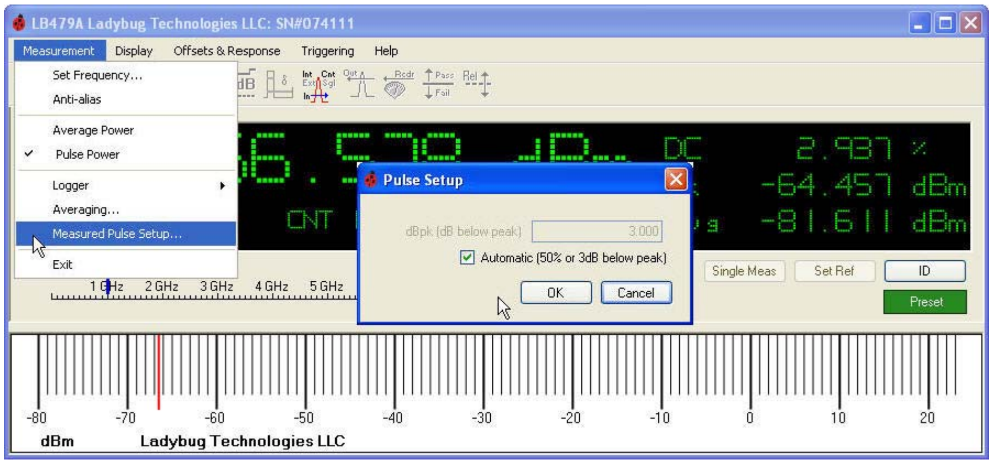
```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetAutoPulseEnabled(
      int addr,
      FEATURE_STATE state );

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

Get(Set)PulseCriteria

Description:

These routines set 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 value you have chosen. For instance, if you set the criteria to 6dB and turn the auto criteria on and off, you will be toggling between 3 and 6 dB below peak. This can also give you some sense of rise time or slope and the sensitivity to this criteria.


LB479A Ladybug Technologies LLC: SN#074111

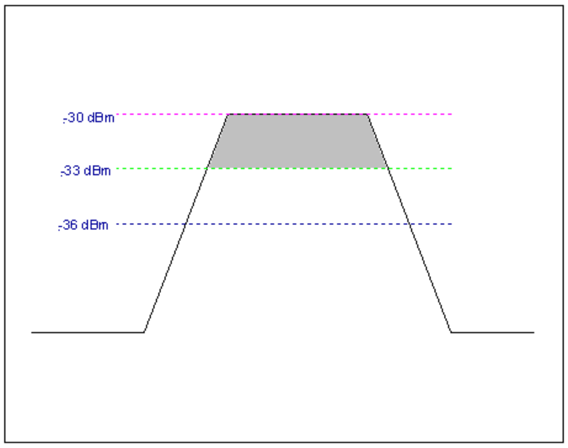
Pulse Setup

dBpk (dB below peak) 3.000

☑ Automatic (50% or 3dB below peak)

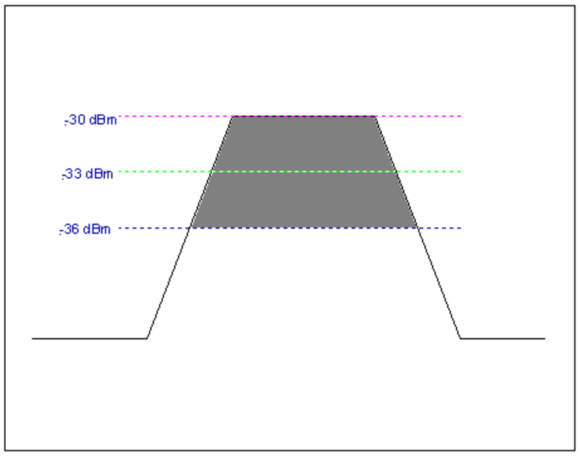
NOTE: Pulse criteria is assumed to be specifed as “dB below peak.” If the peak value is -30dBm and the criteria is 3dB, then the pulse criteria will be -33dB during the measurement. Normally you’ll specify a positive value. Likewise, if you chose 6dB you would be using -36dB. As long as your overshoot is minimal and rise time is relatively steep, the automatic criteria shown above is adequate for most applications.

The diagrams below are intended to help clarify pulse criteria.



A. Three dB Pulse Criteria

In diagram A, 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.



B. Six dB Pulse Criteria

In diagram B, the peak value remains -30dBm and 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. You can see that the average in diagram B will be lower than the average in diagram A.

Pass Parameters:

Address, value of the peak criteria in dB below peak

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)Offset(Enabled)

Description:

These routines 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. If you need 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.

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)DutyCycleEnabled(DutyCyclePerCent)

Description:

CW power sensors have traditionally “adjusted” their power reading based on a known duty cycle provided by the user. Since the Ladybug sensors can measure pulse power and duty cycle directly this is generally viewed as an obsolete technique. However, this feature has been included in deference to this tradition and for comparative reasons.

The calculation to adjust for duty cycle is:

$10\mathrm{Log}_{10}(\text{Duty Cycle})$

So if your duty cycle was assumed to be 10%, the calculation for equivalant average power would be:

$10\mathrm{Log}_{10}(\ 0.1) = -10\ \mathrm{dB}$

This means the average power of a signal with a 10% duty cycle will be 10 dB below the peak value. For meters 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.

The functions provided in the API can enable or disable the feature and set the per cent 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 per cent.

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)CWReference

Description:

These functions setup the sensor for relative measurements during CW measurements. There are separate functions specific to setting the reference for pulse measurements.

NOTE: To make relative measurements you must set the units of measure to “dB Relative”.

Also look at the functions LB_SetMeasurementPowerUnits and LB_GetMeasurementPowerUnits for more information. You may change the reference while in a relative measurement. All relative measurements are made as a ratio and reported as dB above or below the reference.

Related functions:

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)
```

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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(
      int addr,
      ref double relRef,
      ref PWR_UNITS units );
```

Set(Get)PulseReference

Description:

These functions configure the sensor for relative measurements during pulse masurements. There are separate functions specific to setting a reference for CW measurements.

NOTE: To make relative measurements you must set the units of measure to "dB Relative".

Also look at the functions LB_SetMeasurementPowerUnits and LB_GetMeasurementPowerUnits for more information. You may change the reference while in a relative measurement. All relative measurements are made as a ratio and reported as dB above or below the reference.

Related functions:

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
```

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)MeasurementPowerUnits

Description:

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

```
DBM = 0      ' dBm
DBW = 1      ' dBW
DBKW = 2     ' 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 sensor is measuring CW the CW reference is used. When making pulse measurements the pulse reference values are used as the basis for the measurements.

Related functions:

LB_SetCWReference
LB_GetCWReference
LB_SetPulseReference
LB_GetPulseReference

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

Return Value:
Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)TTLTriggerInEnabled(TTLTriggerInInverted)(TTLTriggerInTimeOut)

Description:

These functions control or read back the state of the external trigger input. The trigger-in features are available on sensors 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 time out 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.

Related Functions:

LB_MeasureCW
LB_MeasureCW_PF
LB_MeasurePulse
LB_MeasurePulse_PF

Pass Parameters:

None

Return Value:
Success: > 0
Error: <= 0

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 "C:\LB_APPS\LB_API2\Debug\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 SHARP

```
[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 );
```

Set(Get)TTLTriggerOutEnabled(TTLTriggerOutInverted)

Description:

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

These functions 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 then 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. Then after a few microseconds it will return to a TTL high.

Pass Parameters:

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

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)AntiAliasingEnabled

Description:

Normally, the sampling rate is 500 kHz. As the baseband signals approach the Nyquist criteria (realistically about 200 kHz in this case) problems arise. There are several approaches that can be used to resolve this problem. We use an anti-aliasing capability that really amounts to randomizing 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, if you are measuring signals that have baseband content greater than about 200 kHz we recommend turning on the anti-aliasing feature. These functions enable or disable this feature or allow you to check its state.

Pass Parameters:

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

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Set(Get)RecorderOutEnabled(RecorderOutSetup)

Description:

These features only apply to those instruments with the recorder output option.

Recorder is a 0 VDC to 1 VDC output that is proportional to the measured power. The relationship of the limits and power levels are determined by the user. The recorder output is passed through a 10 Hz hardware filter. Note that recorder-out and trigger-in/out functionality are mutually exclusive.

The recorder output is updated each time the power is measured so that the rate of update is tied to your rate of measurement. To increase the rate of updates perform the following:

- Reduce the number of averages
- Increase your rate of measurement
- Run as few processes as possible on the PC that is being used
- Minimize other tasks

The setup for the recorder out is fairly simple. You need to specifiy the address of the sensor; determine a power level for the 0 V reading; determine a power level for the 1 V reading; and specify the units of measure. Units of measure serve to specify the power levels *and* they are used to specify the scale for interpolation.

So if the units are logarithmic (dBm, dBuV, etc.), the interpolation between 0 VDC and 1 VDC will be with respect to that scale. Likewise, if the units are Watts or Volts the interpolation will be with respect to that scale (Watts or Volts).

For example, if you set the 0 VDC level to -30 dBm and the 1 VDC level to -20 dBm then -25 dBm would produce 0.5 VDC. And if you set the 0 VDC level to 1 mW and the 1 VDC level to 7 mW, then a reading of 4 mW would produce 0.5 VDC. Note that the algorithm will accept any relationship between the 0 and 1 volt levels so that you could have the following:

0 VDC level = -10 dBm
1 VDC level = -30 dBm

In this case -20 dBm would produce a voltage of 0.5 VDC.

Setting up the recorder output is independent of enabling or disabling the feature. The measurement units are:

```
DBM = 0      ' dBm
DBW = 1      ' dBW
DBKW = 2     ' dBkW
DBUV = 3     ' dBuV
W = 4        ' Watts
V = 5        ' Volts
DBREL = 6    ' dB Relative
```

Pass Parameters:

Address, 0 VDC power level, 1 VDC power level, power units

OR

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

Return Value:

Success: > 0
Error: <= 0

Declarations:

C++

```
LB_API2 long _stdcall LB_SetRecorderOutSetup(
      long addr,
      double val_0_V,
      double val_1_V,
      PWR_UNITS units);

LB_API2 long _stdcall LB_GetRecorderOutSetup(
      long addr,
      double* val_0_V,
      double* val_1_V,
      PWR_UNITS* units);

LB_API2 long _stdcall LB_GetRecorderOutSetup(
      long addr,
      double* val_0_V,
      double* val_1_V,
      PWR_UNITS* units);

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

VB 6.0

```
Public Declare Function LB_SetRecorderOutSetup _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByVal val_0_V As Double, _
      ByVal val_1_V As Double, _
      ByVal units As PWR_UNITS) _
      As Long

Public Declare Function LB_GetRecorderOutSetup _
      Lib "LB_API2.dll" ( _
      ByVal addr As Long, _
      ByRef val_0_V As Double, _
      ByRef val_1_V As Double, _
      ByRef units As PWR_UNITS) _
      As Long

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

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

VB.NET

```
Public Declare Function LB_SetRecorderOutSetup _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal val_0_V As Double, _
      ByVal val_1_V As Double, _
      ByVal units As PWR_UNITS) _
      As Integer
Public Declare Function LB_GetRecorderOutSetup _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByRef val_0_V As Double, _
      ByRef val_1_V As Double, _
      ByRef units As PWR_UNITS) _
      As Integer
Public Declare Function LB_SetRecorderOutEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByVal st As FEATURE_STATE) _
      As Integer
Public Declare Function LB_GetRecorderOutEnabled _
      Lib "LB_API2.dll" ( _
      ByVal addr As Integer, _
      ByRef st As FEATURE_STATE) _
      As Integer
```

C SHARP

```
[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_SetRecorderOutSetup(
      int addr,
      double val_0_V,
      double val_1_V,
      PWR_UNITS units );

[System.Runtime.InteropServices.DllImport("LB_API2.dll")]
public static extern int LB_GetRecorderOutSetup(
      int addr,
      ref double val_0_V,
      ref double val_1_V,
      ref PWR_UNITS units );

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

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

Set(Get)ResponseEnabled(Response)

Description:

Response is a frequency sensitive offset, so as you change the measurement frequency 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 you must also specify the number of points.

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)

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
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 );
```

Set(Get)LimitEnabled(SingleSidedLimit)(DoubleSidedLimit)

Description:

Limits are fixed values against which a measured value is compared. Then an evaluation is expressed as pass or fail (normally). This evaluation is done during a measurement. Specifically, the evaluation is made and returned during either LB_MeasureCW_PF or LB_MeasurePulse_PF.

The limits come in two types: The first type is a single line. The value can be below, equal to, or above this line. You may specify any of these conditions as pass or fail.

NOTE: By specifying either pass or fail you have implied the other. The convention used by Ladybug is to specify the passing condition. Failing is implied by not passing.

The second type of limit is a 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 the limit:

- Address (sensor 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 specifiying that neither limit is enabled, single-sided limits are enabled or double-sided limits are enabled.

NOTE: In the declarations section below some of the comments have been truncated for brevity. Also, we have shown the enumerations specific to limits with the exception of units. Units 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)

Return Value:

Success: > 0
Error: <= 0

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 measured value is greater than the
      lower limit and
                              // less than the upper limit
      PASS_BETWEEN_INC  = 1,  // Pass if measured value is equal to or greater
                              // limit and equal to or less than the upper
      PASS_OUTSIDE_EXC  = 2,  // Pass if measured value is less than the lower
                              // greater than the upper limit
      PASS_OUTSIDE_INC  = 3   // Pass if measured value is equal to or greater
                              // 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 but
                        // not limited to 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 = 5        ' no determination made possible reasons
                                 ' 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_SetLimitEnabled _
      Lib "LB_API2.dll" ( _
      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        ' no determination made possible reasons
                              '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_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 SHARP

```
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 );
```

Set(Get)CalDueDate

Description:

This feature is for your use. While the calibration due date is set at the factory, you are free to change it as your unit gets calibrated. More importantly, your systems can interrogate the sensor for its cal due date. The one element that is important to note is that this is specified by *serial number, not address*.

Pass Parameters:

NA

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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);
```

Store(Recall)Reg

Description:

These functions are the traditional store/recall register functions. 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 Ladybug application directory.

This means that saved states can be saved, copied or moved between PCs. Any sensor 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:

LB478_xxx.INI
LB479_xxx.INI
LB480_xxx.INI

Where xxx is the address of the unit. More importantly, the names of the files may be renamed and managed by your application as you see fit (perhaps stored in a data base as a long string). This means that the store/recall registers are now under your complete control.

Pass Parameters: None

Return Value:

Success: > 0
Error: <= 0

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 SHARP

```
[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 );
```

Factory Function Calls Listing

LB_GetCalOptExpDate

LB_GetWtyOptExpDate

LB_GetConnectorOption

LB_GetCalAndWtyOption

LB_GetRecorderOutOption

LB_GetBestMatchOpt

LB_GetTriggerOpt

LB_GetFilterOpt

Description:

These calls are intended primarily for the factory. However, they may have some use to others and so we have included them in the declaration files. Each of the “Get” functions has a companion “Set” function. The “Set “ functions require a password available to the factory only. Rather than detail calls here we want to point out that they are available if required. If additional information is required please contact Ladybug Technologies LLC.